UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

DESARROLLO DE UNA APLICACIÓN ITS:
detección de colisión

Trabajo de fin de grado

Presentado en la facultad de

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

por

Sergio Liarte Genovés

En cumplimiento parcial

De los requerimientos para el grado de

INGENIERÍA TELEMÁTICA

Tutor: Jordi Casademont

Barcelona, Junio 2019

Abstract

The project presented in this paper deals with the creation of a C-ITS application, a technology that wirelessly communicates terrestrial transports, to warn of possible collisions with other vehicles. It is a first version of an application that uses existing standards and open code programs that implement it. As a result it have been obtained an application that works correctly in some scenarios but that has to improve the performance of time to implement it in the real world.

Resum

El projecte presentat en aquest treball tracta la creació d'una aplicació C-ITS, tecnologia que comunica sense fils els transports terrestres, per avisar de posibles col·lisions amb altres vehicles. És una primera versió d'una aplicació que utilitza estandards existents i programes de codi lliure que ho implementen. Com a resultat s'ha obtingut una aplicació que funciona correctament en alguns escenaris però que ha de millorar les prestacions de temps per implementar-ho al món real.

Resumen

El proyecto presentado en este trabajo trata la creación de una aplicación C-ITS, tecnología que comunica inalámbricamente los transportes terrestres, para avisar de posibles colisiones con otros vehículos. Es una primera versión de una aplicación que utiliza estándares existentes y programas de código libre que lo implementan. Como resultado se ha obtenido una aplicación que funciona correctamente en algunos escenarios pero que ha de mejorar las prestaciones de tiempo para implementarlo en el mundo real.

Agradecimientos

Parcialmente financiado por ”La Secretaria d'Universitats i Recerca del Departament d'Empresa i Coneixement de la Generalitat de Catalunya” a través del proyecto 2017 SGR 00376.

Historial de revisiones y registro de aprobación

Revisión	Fecha	Propósito
0	06/06/2019	Creación del documento
1	23/06/2019	Primera revisión del documento
2	24/06/2019	Segunda revisión del documento

LISTA DE DISTRIBUCIÓN DEL DOCUMENTO

Nombre	e-mail
Sergio Liarte	liarte97@gmail.com
Jordi Casademont	jordi.casademont@upc.edu

Escrito por:		Revisado y aprobado por:	
Fecha	23/06/2019	Fecha	25/06/2019
Nombre	Sergio Liarte	Nombre	Jordi Casademont
Rango	Autor del proyecto	Rango	Supervisor del proyecto

Índice

Lista de Figuras

Lista de Tablas

1. Introducción

Este proyecto trata la creación de una aplicación Cooperative Intelligent Transport System (C-ITS), una tecnología que trata de comunicar los vehículos terrestres para mejorar su seguridad y confort. Para realizar el proyecto se ha investigado estándares de implementación de esta tecnología, en tecnologías GPS y se ha utilizado el lenguaje C++ para programar la aplicación principal y Javascript para crear una página web de comprobación.

La aplicación recibe mensajes que envían el resto de vehículos y los procesa para detectar si puede haber alguna colisión con otro vehículo. Acto seguido, avisa al usuario del vehículo de estas colisiones que pueden producirse si no toma precauciones. De esta manera, se pretende evitar colisiones frontales de vehículos que circulan en la misma dirección o colisiones en intersecciones con poca visibilidad.

Como resultado, se ha obtenido una aplicación que funciona correctamente advirtiendo de las posibles colisiones cuando se desea. Pero para implementarla en el mundo real aún falta mucho recorrido. Se necesita conseguir darle a la aplicación más precisión en los datos GPS para detectar todo más claramente y se necesitaría mejorar el rendimiento a la hora de procesar mensajes para no saturarse en algunos escenarios.

		PT 1		PT2			PT3			PT4			PT5				PT6			PT7	PT8	PT9			PT10	
		T1: lectura	T2: resumenes	T1: instalar SO	T2 instalar libreñas	T3 instalar Vanetza	T1: enviar paquetes	T2: capturar mensajes Wireshark	T3: decodificar mensajes recibidos	T1: leer información GPS	T2: investigar unidades medida	T3: relacionar GPS, velocidad, dirección	T1: coger información de mensajes	T2: introducir información en la tabla	T3: mantenimiento tabla	T4: obtención de información de la tabla	T1: diagrama flujo	T2: creación algoritmo	T3: implemntación diagrama y algoritmo	T1: integración todas las partes	T1: solucionarheading/speed CAM	T1: propuesta de trabajo y plan de trabajao	T2: revisión crítica	T3: memoria	T1: preparación defensa	T2: defensa TFG
Enero	S1(28-01)																									
	S2 (04-02)																									
Febrero	S3 (11-02)																									
	S4 (18-02)																									
	S5 (25-02)																									
Marzo	S6 (04-03)																									
	S7 (11-03)																									
	S8 (18-03)																									
	S9 (25-03)																									
Abril	S10 (01-04)																									
	S11(08-04)																									
	S12 (22-04)																									
	S13 (29-04)																									
Mayo	S14 (06-05)																									
	S15 (13-05)																									
	S16 (20-05)																									
	S17 (27-05)																									
Junio	S18 (03-06)																									
	S19 (10-06)																									
	S20 (17-06)																									
	S21(25-06)																									
Julio	S22 (01-07)																									

Tabla 1: Diagrama Gantt

2. Estado del arte

2.1. Cooperative Intelligent Transport System

Recientemente ha surgido el concepto de Sistemas Inteligentes de Transporte (ITS) que al ser algo nuevo tiene muchas definiciones y van cambiando constantemente. Pero se podrían resumir como un conjunto de soluciones tecnológicas telemáticas que surgen con el objetivo de mejorar la seguridad, la fluidez y la economía del transporte terrestre. De hecho, la definición de ITS según la Sociedad Americana del Transporte Inteligente es "gente usando tecnología en transportes para salvar vidas, tiempo y dinero".

Estas tecnologías han surgido porque en los últimos años el tráfico de vehículos terrestres ha incrementado mucho a causa del crecimiento de la población a nivel mundial. Esto, también ha producido que las infraestructuras viarias se vayan quedando atrás y el tráfico se congestione más incrementando así el tiempo de viaje, el consumo de combustible y la contaminación ambiental.

Algunas aplicaciones prácticas de ITS son visibles ya en algunos lugares como pueden ser los sistemas de tele peaje sin parar el vehículo, sistemas de llamada de emergencia desde el coche en caso de accidente, semáforos inteligentes en función del tráfico o sistemas para detectar infracciones como saltarse semáforos en rojo o circular por el carril bus.

Dentro de los ITS se puede separar un subconjunto de tecnologías que se dedican a comunicarse entre diferentes ITS inalámbricamente, los Cooperative Intelligent Transport System (C-ITS).

2.1.1. Arquitectura en capas

Actualmente hay 5 arquitecturas distintas sobre C-ITS: una europea (diseñada por la ETSI); una internacional pero adoptada mayoritariamente en Europa (diseñada por CEN/ISO); una norteamericana (diseñada por IEEE), una japonesa (diseñada por JSAE) y una coreana (diseñada por ITS Korea).

En la figura 1 se puede ver la arquitectura que ha adaptado la ETSI para este tipo de tecnologías.

Se pueden observar 4 capas horizontales (acceso al medio, red y transporte, facilidades y aplicación) y 2 verticales que interactúan con las horizontales (gestión y seguridad). Más detalle acerca de las capas en ANEXO 1.

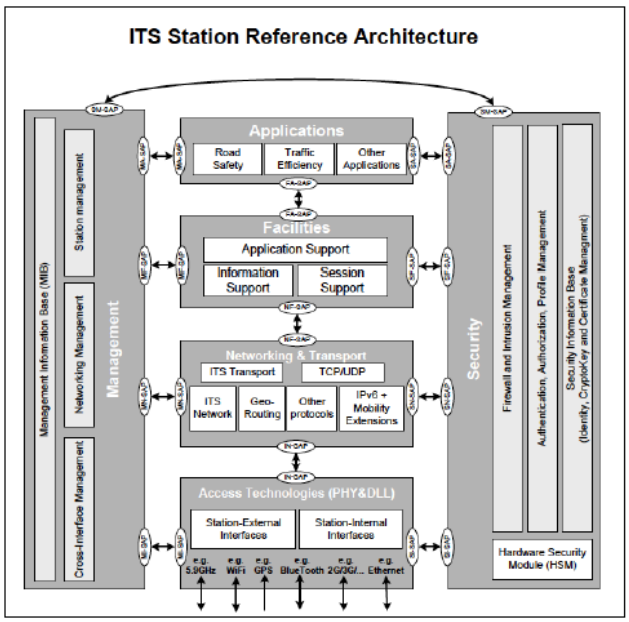

Figura 1: Arquitectura en capas de un sistema ITS [4]

2.1.2. Arquitectura del sistema

Los sistemas C-ITS tienen básicamente 4 elementos principales que requieren conexiones entre ellos:

- **Vehicular ITS station**: Dispositivo instalado en los vehículos. El equipo para implementar estas estaciones se denomina On Board Unit (OBU).
- **Personal ITS station**: Smartphones y otros dispositivos cuyo objetivo sea asistir a usuarios de vehículos u otros usuarios vulnerables (ciclistas, peatones, ...).
- **Roadside ITS station**: Infraestructura vial, dispositivos instalados al lado de la vía. El equipo para implementar estas estaciones se denomina Road Side Unit (RSU).
- **Central ITS station**: Centros de gestión de tráfico y los back-office de los proveedores de servicios.

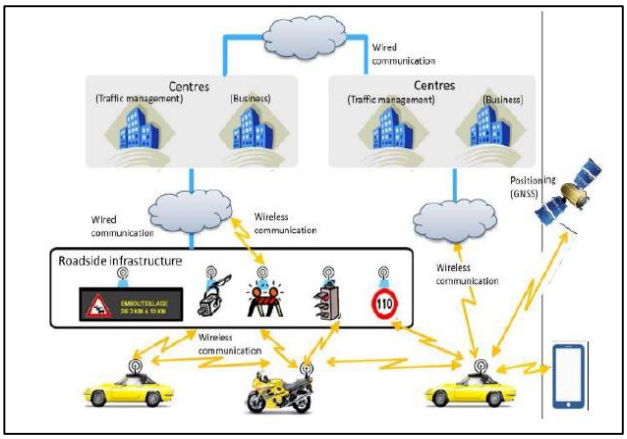

Figura 2: Esquema de elementos de un sistema C-ITS [4]

Las conexiones entre elementos se clasifican de diferente manera atendiendo a sus emisores y receptores:

- **V2V**: Vehículo a vehículo.
- **I2I / R2R**: Infraestructura a infraestructura / Infraestructura vial (Roadside) a infraestructura vial.
- **V2I o I2V / V2R o R2V**: Vehículo a infraestructura o infraestructura a vehículo / Vehículo a infraestructura vial (Roadside) o infraestructura vial a vehículo.
- **V2C o C2V**: Vehículo a centro o centro a vehículo.
- **I2C o C2I / R2C o C2R**: Infraestructura a centro o centro a infraestructura / Infraestructura vial (Roadside) a centro o centro a infraestructura vial.
- **V2P o P2V**: Vehículo a persona o persona a vehículo. Dentro de la categoría persona encontramos a peatones, ciclistas, conductores o pasajeros que puedan tener acceso a algún terminal inteligente.

Las arquitecturas V2V consisten en comunicaciones directas entre vehículos cercanos. Tienen la ventaja del mínimo retardo, pero tienen en contra que son susceptibles a congestionarse en escenarios con muchos vehículos.

Hay dos tipos de arquitecturas V2I: centralizadas y distribuidas.

Las arquitecturas **centralizadas** suelen ser combinaciones de V2I y I2C. Generalmente, utilizan tecnologías de largo alcance (celulares). Tienen la ventaja de la capacidad de proceso del Cloud centralizado y su escalabilidad, pero el inconveniente del retardo que introducen las comunicaciones entre vehículos.

En las arquitecturas **distribuidas**, los vehículos se comunican con infraestructuras cercanas a las carreteras, Road Side Units (RSUs) que tienen más capacidad de proceso que las On Board Unit (OBUs) que se encuentran en el vehículo. Además, tienen una visión más general al comunicarse con todos los vehículos.

2.1.3. Evolución

Una idea de hacía donde va esta tecnología la tenemos en la figura 3:

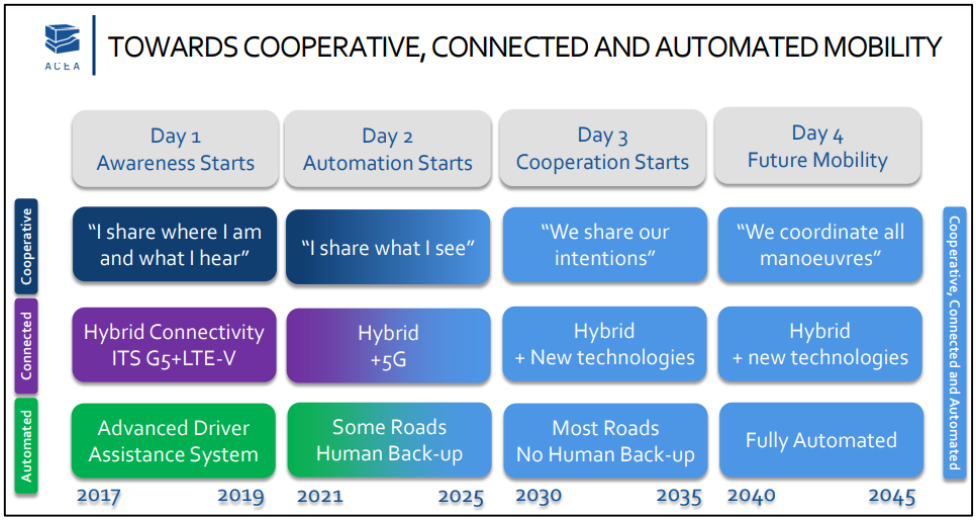

Figura 3: Cuadro predictivo de las tecnologías C-ITS [6]

Siguiendo con la figura, antes del día 1, había un día 0. Anteriormente, y en la actualidad, los coches ya llevaban tecnología incorporada, aunque no se comunicaban entre ellos. Pueden detectar obstáculos en su trayectoria a través de algunos sensores; incorporan la adaptación de la velocidad crucero a los coches que les preceden; tienen avisos y ayuda en la detección de líneas de detención o cambio de carril; ayudan a aparcar o incluso aparcan solos.

En el día 1 suponemos que algunos coches ya comparten información: posición, velocidad, y eventos varios. De esta manera ya se pueden crear aplicaciones que informen de si hay mucho tráfico o hay obras en la calzada en una zona y así poder trazar una ruta alternativa; pueden avisar al conductor de una posible colisión en un cruce sin visibilidad; pueden avisar al conductor de que se acerca un vehículo de emergencia; o pueden recomendar al conductor la velocidad ideal a la que debe ir para que un semáforo en rojo se cambie a verde al llegar sin tener que frenar (GLOSA).

En el día 2 los vehículos empiezan a compartir con el resto de vehículos la información de sus sensores para que así el resto de vehículos pueda procesar información de los vehículos que no pueden compartir información. De esta manera se puede mejorar servicio de información de obras en la carretera ofrecido por los propios usuarios; se pueden crear ayudas para el adelantamiento de vehículos; se puede automatizar la adaptación de la velocidad crucero a través de mensajes y no con sensores…

Una vez los vehículos ya tienen información de todo su alrededor, el objetivo del día 3 es conducir de una manera más eficiente entre todos. Para ello se piensa en aplicaciones como la sugerencia de ruta para adelantar en función de la velocidad del resto; el aviso de usuarios vulnerables en la vía; el GLOSA 2.0, que incluye en la recomendación la velocidad del resto de coches hasta llegar al semáforo; o el platooning (un conjunto de camiones circulando solos por la carretera detrás de un camión conducido por una persona para así ahorrar gastos referente al consumo de combustible) pero para ello se necesita mucha comunicación entre camiones y entre otros vehículos que quieran adelantar, entrar o salir de la vía.

Y al final del camino estaría la conducción totalmente autónoma donde los coches adelantarían solos, regularizarían entre ellos las intersecciones… y los humanos tan solo mirarían como los coches circulan solos.

2.2. Vanetza

Como se puede ver, en ITS hay muchos avances y ya hay estándares hechos. Por tanto, también hay varios softwares que implementan aplicaciones de este tipo. Pero la gran mayoría son privados e inaccesibles. Para la realización de este proyecto se ha utilizado la librería Vanetza [7].

Esta librería de software libre accesible en GitHub está programada en C++. Contiene gran cantidad de información para simular el protocolo de la ETSI C-ITS. Contiene varios protocolos ya preparados para simular:

- GeoNetworking.
- Basic Transport Protocol.
- Decentralized Congestion Control.
- Seguridad.
- Algunos mensajes extraídos del ASN.1 como CAM i DENM.

Las carpetas principales de vanetza y que dan una idea de todo lo que abarca son: access, asio, asn1, btp, common, dcc, facilities, geonet, gnss, net, security y units.

Pero Vanetza es solo la librería que da soporte en los 2 niveles inferiores (acceso, red/transporte) de la aqruitectura ETSI y parte del nivel 3 (facilidades) . Vanetza permite desarrollar proyectos sobre la capa de aplicación olvidándote de las capas inferiores.

En 2016, un conjunto de estudiantes crearon la herramienta Socktap. Socktap son un conjunto de programas que gracias a la librería Vanetza, corren un programa que envía paquetes C-ITS. Hay creados tres programas:

- hello: Envía un paquete sobre btp con un payload hardcode.

- cam: Envia un mensaje CAM con unos valores modificables en el programa.

- bench_in: Envía mensajes y te muestra por pantalla cuando ha recibido uno.

Esta herramienta ya incluye certificados de seguridad, recibe la posición GPS del daemon gpsd, y funciona en las diferentes interfaces de red del ordenador.

2.3. GPS

Gpsd es un daemon de software libre que recibe la información de un receptor GPS y la pone a disposición de otras aplicaciones en el puerto TCP 2947.

Para facilitar la simulación, no es cómodo conectar un GPS y trabajar con datos reales. Así que también existe el programa gpsfake. Este programa simula el receptor GPS a través de un archivo de texto con su correspondiente formato.

Gpsfake abre una pseudo consola que lanza una instancia gpsd que piensa que el lado esclavo de la consola es su dispositivo GPS y la alimenta con datos de un archivo. Se le pueden pasar varios archivos que irá intercalando en el orden deseado y se le pueden pasar diferentes opciones relacionadas con intervalo de repetición, puerto de salida, monitorización, logs…

3. Desarrollo del proyecto

3.1. Tecnologías

3.1.1. Cooperative Awareness Message (CAM)

Los CAM [1][2] son los mensajes en los que se basa la aplicación. Pueden incorporar mucha información dependiendo del vehículo de donde procede. Pero lo importante para la aplicación son el identificador de la estación origen, la marca de tiempo, la posición geográfica de la estación, su dirección y velocidad. La frecuencia de los mensajes CAM es de 10 cada segundo.

A continuación, se explica cómo están formados los mensajes y como están representados los datos que nos interesan para la aplicación.

Un CAM está formado por una cabecera ITS PDU y múltiples contenedores.

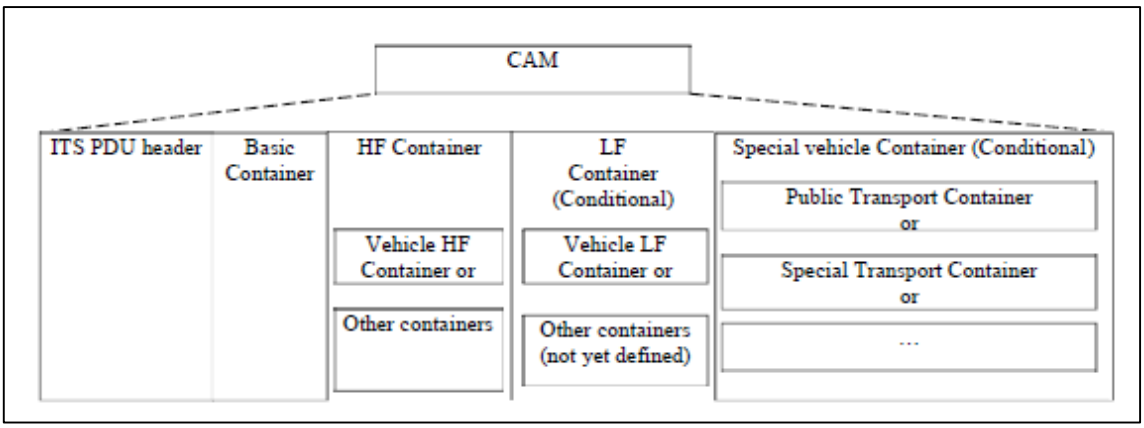

Figura 4: Esquema de un mensaje CAM

ITS PDU header: contiene la versión del protocolo, el tipo de mensaje y el ID de la estación ITS de dónde procede el mensaje.

Basic container: contiene el tipo de la estación ITS de dónde procede el mensaje y la última posición geográfica de dicha estación.

Todos los CAMs generados por vehículos ITS deben contener al menos el contenedor *high frecuency vehicle* (Vehicle HF) y, opcionalmente, un contenedor *low frequency vehicle* (vehicle LF). El primero contiene información que cambia rápido en el tiempo, como la velocidad o la dirección; y el segundo contiene información que cambia con menos frecuencia, como el estado de las luces del vehículo.

Además, los vehículos que tienen un rol específico en la circulación (transporte público o vehículos de emergencia, por ejemplo), aportan más información en contenedores especiales. Este rol está especificado en un campo del Vehicle LF y hay siete tipos: *publicTranport*, *specialTransport*, *dangerousGoods*, *roadWork*, *rescue*, *emergency* o *safetyCar*.

Los datos que nos interesan son:

generationDeltaTime:

Es el tiempo correspondiente a cuando se tomó la posición de referencia que incluye en mensaje CAM. El valor está en milisegundos, y el valor máximo es 65535.

Para conseguir el tiempo, la estación origen coge el tiempo en milisegundos desde 2004-01-01 T00:00:00:000Z módulo 65536.

Latitude, longitude:

Los dos valores los contiene la estructura *ReferencePosition* que se especifican a continuación:

ReferencePosition ::= SEQUENCE {

latitude Latitude,

longitude Longitude,

positionConfidenceEllipse PosConfidenceEllipse ,

altitude Altitude

}

Todos los valores recogidos en la estructura están tomados en el tiempo indicado en el campo anterior *generationDeltaTime*.

Si el valor del tipo de estación originaria está entre 3 y 11 el punto de referencia debe ser el suelo del centro de masas de la parte frontal del vehículo.

Latitude ::= INTEGER {oneMicrodegreeNorth (10), oneMicrodegreeSouth (-10), unavailable(900000001) } (-900000000..900000001)

Es la latitud geográfica absoluta en un sistema de coordenadas WGS84, que proporciona un rango de 90 grados en el norte o en el hemisferio sur.

Los valores positivos se utilizan para los que están al norte del ecuador y los negativos para los que están al sur del ecuador. Cuando la información no está disponible, debe utilizarse el valor 900 000 001.

La unidad es 0.1 microgrados, por lo tanto, un grado al norte es 10.

Longitude ::= INTEGER {oneMicrodegreeEast (10), oneMicrodegreeWest (- 10), unavailable(1800000001) } (-1800000000..1800000001)

Es la longitud geográfica absoluta en un sistema de coordenadas WGS84, que proporciona un rango de 180 grados al este o al oeste del primer meridiano.

Los valores positivos se utilizan para los que están al este y los negativos para los que están al oeste. Cuando la información no está disponible, debe utilizarse el valor 1 800 000 001.

La unidad es 0.1 microgrados, por lo tanto, un grado al oeste es -10.

PosConfidenceEllipse ::= SEQUENCE {

semiMajorConfidence SemiAxisLength,

semiMinorConfidence SemiAxisLength,

semiMajorOrientation HeadingValue

}

El *positionConfidenceEllipse* proporciona la precisión de la posición medida con un nivel de confianza del 95%. De lo contrario, *positionConfidenceEllipse* se deberá configurar como no disponible.

Si *semiMajorOrientation* se establece en 0° Norte, entonces *semiMajorConfidence* corresponde a la precisión de la posición en la dirección Norte / Sur, mientras que *semiMinorConfidence* corresponde a la precisión de la posición en dirección Este / Oeste. Esta definición implica que la *semiMajorConfidence* podría ser más pequeña que el *semiMinorConfidence*.

Heading

Heading ::= SEQUENCE {

headingValue HeadingValue,

headingConfidence HeadingConfidence

}

HeadingValue ::= INTEGER {wgs84North(0), wgs84East(900), wgs84South(1800), wgs84West(2700), unavailable(3601)} (0..3601)

HeadingConfidence ::= INTEGER {equalOrWithinZeroPointOneDegree (1), equalOrWithinOneDegree (10), outOfRange(126), unavailable(127)} (1..127)

Headingvalue indica la orientación respecto al WGS84 norte. Si la información no está disponible, debe ser 3601.

La unidad de medida es de 0.1 grado. Por tanto, 45º será indicado como 450.

El valor *headingConfidence* proporciona la precisión de la medida con un nivel de confianza del 95%. De lo contrario, el valor de *headingConfidence* se establecerá como no disponible (127).

La unidad es de 0.1 grados. Si es menor a 0.1 grados, se pondrá 1. Si es igual o menos a 12.5 se pondrá 125. Si es mayor a 12.5, 126.

Speed

Speed ::= SEQUENCE {

speedValue,

speedConfidence

}

SpeedValue ::= INTEGER {standstill(0), oneCentimeterPerSec(1), unavailable(16383)} (0..16383)

SpeedConfidence ::= INTEGER {equalOrWithinOneCentimeterPerSec(1), equalOrWithinOneMeterPerSec(100), outOfRange(126), unavailable(127)} (1..127)

Speedvalue indica la velocidad del vehículo. El valor máximo es 163,82 m/s, para valores mayores a este, se pondrá 16 382. Si la información no está disponible, se pondrá 16 383.

La unidad de medida es 0.01 m/s. Por tanto, si es 50.12 m/s se pondrá 5 012.

El valor *speedConfidence* proporciona la precisión de la medida con un nivel de confianza del 95%. De lo contrario, el valor de speedConfidence se establecerá como no disponible (127).

La unidad es de 1 cm/s. Si es menor a 1 cm/s, se pondrá 1. Si es igual o menos a 125 cm/s se pondrá 125. Si es mayor a 125 cm/s, 126.

3.1.2. GPS

Hay muchos formatos [12] en los que el GPS nos indica la posición del receptor y los satélites a los que está accediendo. Para este trabajo se ha podido comprobar que con los siguientes formatos es suficiente para el correcto funcionamiento del programa:

Formato $GPRMC: Recomended Minimum Specific GPS/Transit data

Ejemplo de línea: $GPRMC,081836,A,3751.65,S,14507.36,E,000.0,360.0,130998,011.3,E*62

Los valores que nos interesan son el tercero y cuarto que nos aportan la latitud; el 5 y 6 que nos aportan la longitud el 7 la velocidad y el 8 la dirección.

Más información en ANEXO 2.

Formato $GPGGA: Global Positioning System fix data

Ejemplo de línea: $GPGGA,141107.978,4123.461,N,00209.529,E,1,12,1.0,0.0,M,0.0,M,,*68

Repite información del formato anterior y nos da información sobre satélites y antena.

Más información ANEXO 2.

En la página web https://rl.se/gprmc [9] se puede comprobar si una línea $GPRMC o $GPGGA es correcta. Además, nos muestra en un mapa el punto que indica la línea y decodifica los datos en un cuadro a la derecha.

Formato $GPGSV: GPS Satelites in View

Este formato nos indica el número de satélites de los que recibimos señal y la calidad de la señal. Es un formato diferente a los anteriores ya que se muestra en ciclos y cada ciclo suele incluir más de una línea. Cada línea incluye información de varios satélites. Todas las líneas incluyen información común referente a cada ciclo y después cada una puede incluir hasta un máximo de información de 4 satélites.

Ejemplo de ciclo:

$GPGSV,3,1,12,04,45,152,31,07,06,327,,08,17,286,30,10,35,146,33*7B

$GPGSV,3,2,12,16,82,255,41,20,48,111,45,21,49,051,45,26,59,166,42*7B

Más información en ANEXO 2.

3.2. Desarrollo de la aplicación

El proyecto está pensado según el esquema de la figura 5:

Dos Raspberry Pis conectadas a dos GPS diferentes. Una raspberry hace de emisor utilizando el programa socktap-cam que proporciona Vanetza. Otra Raspberry hace de receptora. Este receptor, recibe los mensajes a través de un parser, una aplicación ya hecha en un trabajo final de máster anterior [8]. Este parser, procesa los valores del mensaje CAM y se los pasa a la aplicación desarrollada en este trabajo.

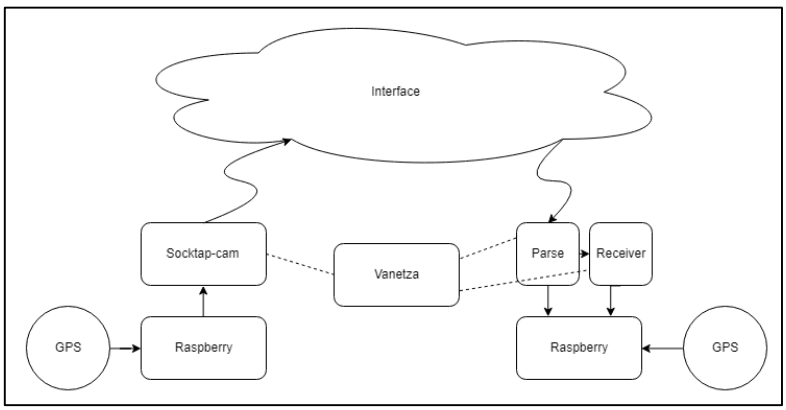

Figura 5: Esquema del escenario de prueba

Este es el escenario de prueba. Pero puede haber más emisores y no es obligatorio que exista un receptor GPS real, se puede simular.

En este proyecto se ha tratado la creación de un programa que funciona dentro de la "caja" del receiver. Antes de explicar el flujo es importante tener la idea de que la aplicación está dividida en 4 partes: distintas entradas que guardan información; distintas tablas que guardan las entradas de manera ordenada; un controlador que organiza todas las tablas; y una clase auxiliar que realiza todas las operaciones matemáticas para determinar distancias o posiciones geográficas.

3.2.1. Flujo de la aplicación

La aplicación funciona siguiendo el flujo que se puede ver en los diagramas 6 i 7.

Cuando llega un mensaje CAM de otro vehículo lo primero que hace la aplicación es comprobar si la dirección del propio vehículo que le proporciona el GPS es fiable. Para ello, evalúa una variable booleana a la que se le da valor al iniciar el programa. En el caso de que sea fiable, se actualiza la dirección con la que le proporciona el GPS y el programa sigue su curso. Si la dirección del GPS no es fiable, se calcula la dirección propia a partir de la posición anterior que tenía el programa guardada y la nueva dirección que le proporciona el GPS. A continuación, actualiza su propia posición (latitud y longitud) y velocidad.

El programa sigue su ciclo comprobando si la dirección de los vecinos que viene indicada en el mensaje CAM es fiable o no. Para ello, igual que con la dirección propia, comprueba una variable booleana que se le da valor al iniciar el programa. En caso de que sea fiable, el programa continúa sin cambiar nada. En caso contrario, entra en acción una de las tablas del programa. Básicamente, el programa intenta conseguir la dirección a partir de posiciones antiguas guardadas y la actualiza en el mensaje propio del programa para utilizarlo en un futuro.

El siguiente paso del programa es comprobar si el ID de la estación que viene especificado en el mensaje CAM lo tiene guardado en la tabla de distancias que mantiene. En caso negativo, crea la entrada con los campos necesarios y la añade a la tabla. En caso de tener guardado el ID, realiza unas operaciones para saber si el vehículo del mensaje se está alejando o acercando a su propia posición. Si se está alejando, no hace nada más con ese mensaje y realiza el paso final del programa. Si se está acercando, entonces el programa calculará el punto y en qué momento en el tiempo se puede producir la colisión a partir de la velocidad que están manteniendo ambos vehículos en ese momento.

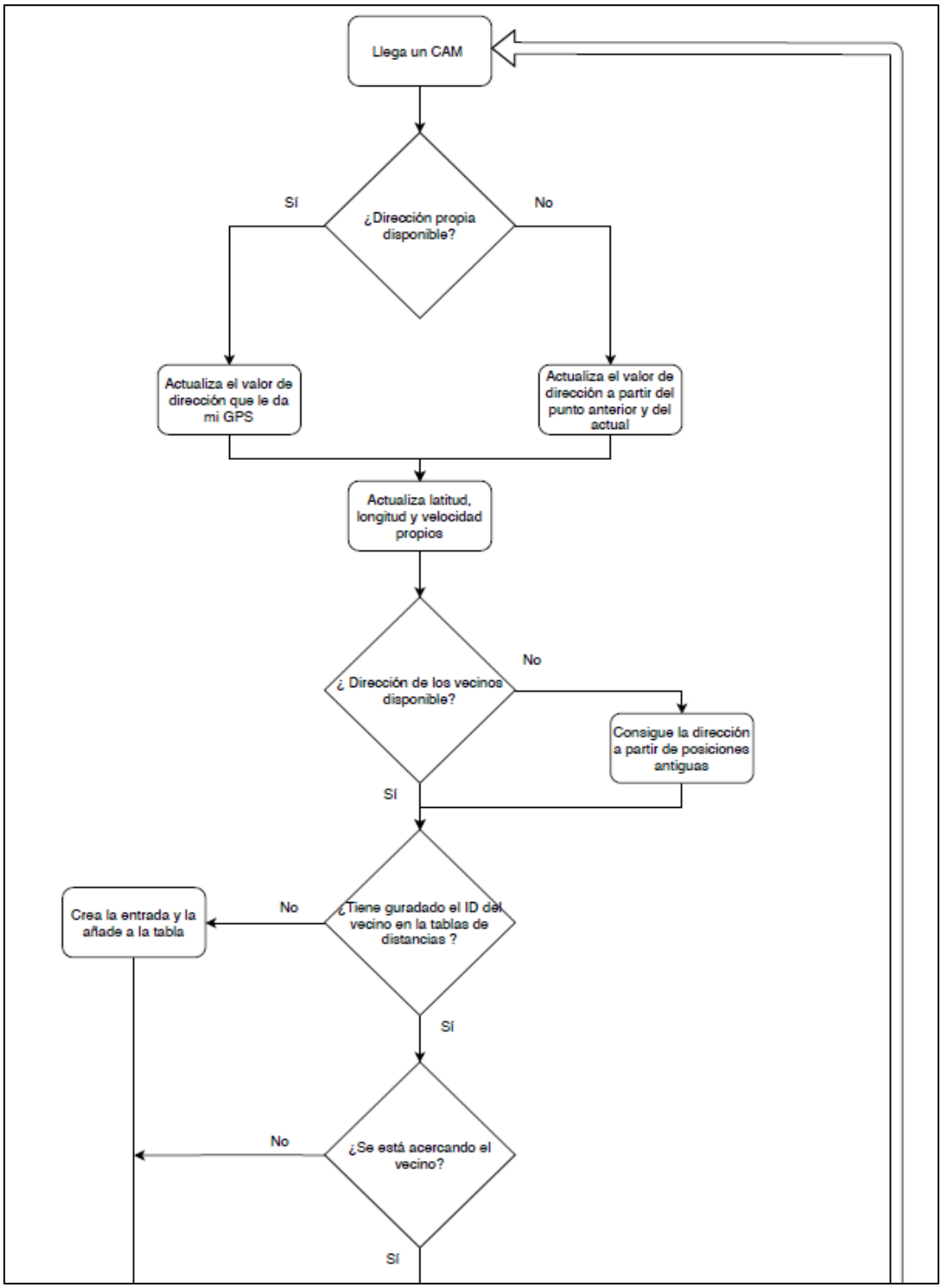

Figura 6: Flujo de la aplicación (1/2)

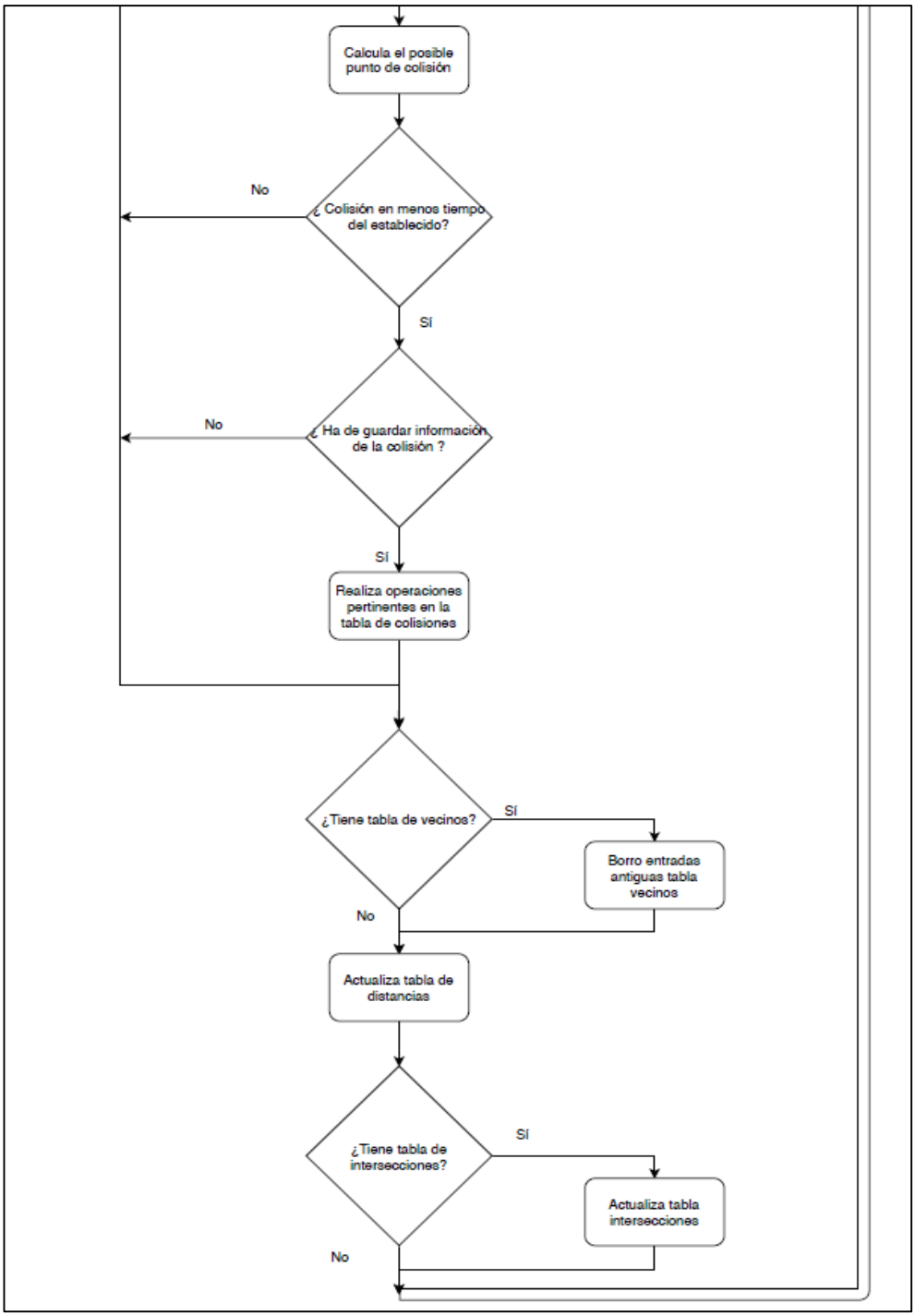

Figura 7: Flujo de la aplicación (2/2)

Una vez tiene calculado el punto de la posible colisión comprueba si el tiempo en el que se producirá es mayor o menor a la variable que tiene guardada para avisar al usuario de la posible colisión o no. Esta variable se configura al inicio del programa y está pensada para no avisar de colisiones que pueden suceder en un lapso de tiempo muy grande donde pueden suceder muchas cosas para evitarlas. En caso de ser mayor, el programa no hace nada más con ese mensaje y realiza el paso final. Si, por el contrario, es menor, quiere decir que se puede producir la colisión si no se hace nada para evitarlo. Así que el programa muestra por pantalla la información de la colisión necesaria para evitarla: la posición (latitud y longitud) y el tiempo que falta para que suceda.

A continuación, el programa comprueba si ha de guardar la información de las posibles colisiones o no. Esta información está pensada para debugar la aplicación o para posibles aplicaciones en un futuro. Pero básicamente es una tabla que guarda la información de las posibles colisiones.
Finalmente, el programa realiza el paso final: borrar las entradas de las tablas porque ya no le serán útiles al ser entradas muy antiguas. Para hacerlo, se rige a unos valores de tiempo que son definidos al iniciarse el programa. Una vez actualizadas las tablas, espera a que vuelva a llegar un CAM para volver a empezar el ciclo.

3.2.2. Control_entries

Las entradas son las clases que utiliza el programa para guardar información en las tablas. Hay de tres tipos: control_entry_neighbour, control_entry_distance, control_entry_intersection. Todas tienen unos atributos diferentes, getters de estos atributos y unos métodos que se utilizan para modificar sus atributos desde las tablas.

3.2.2.1. Control_entry_neighbour

Es la clase básica del programa. Es la que comunica entre el receiver y la aplicación. Cuando el receiver recibe un CAM, crea un control_entry_neighbour y se lo pasa al controlador para que inicie el ciclo de la aplicación.Tiene:

- Atributos:

- **Id** de la estación de donde proviene el mensaje.
- **Tiempo** de creación del mensaje.
- **Latitud** de la posición en formato decimal.
- **Longitud** de la posición en formato decimal.
- **Dirección**/rumbo del vehículo en grados.
- **Velocidad** en metros por segundo.

- Métodos:

- **Getters**: Id, tiempo, latitud, longitud, dirección y velocidad.
- **Print**(): Método que muestra información de la entrada con formato “id: __ || time: ____ || latitude: _____ || longitude: _____ || heading: _____ || speed: _____ “.
- **setData** (tiempo, latitud, longitud, dirección, velocidad): setter de cualquier dato de la entrada.
- **convertPosition** (valor): Este método tan solo multiplica por 10^{-6} y se utiliza en el constructor para la latitud y longitud y así tener la posición en las unidades correctas porque en el mensaje CAM vienen en otro formato. Este método está pensado para guardar las coordenadas en el formato deseado sin tener que tocar mucho el código. Si, por ejemplo, se quiere guardar las coordenadas en unidades sexagesimales, tan solo había que cambiar este método y el programa funcionaría sin tener que cambiar nada más.
- **getRealGenerationDeltaTime**(valor_con_módulo): Este método es necesario para tener el valor de tiempo real que en el que se generó el mensaje CAM. En el mensaje, viene el valor de tiempo en milisegundos, pero con módulo 65536. Para tener el valor de tiempo en milisegundos completo se realiza lo siguiente:

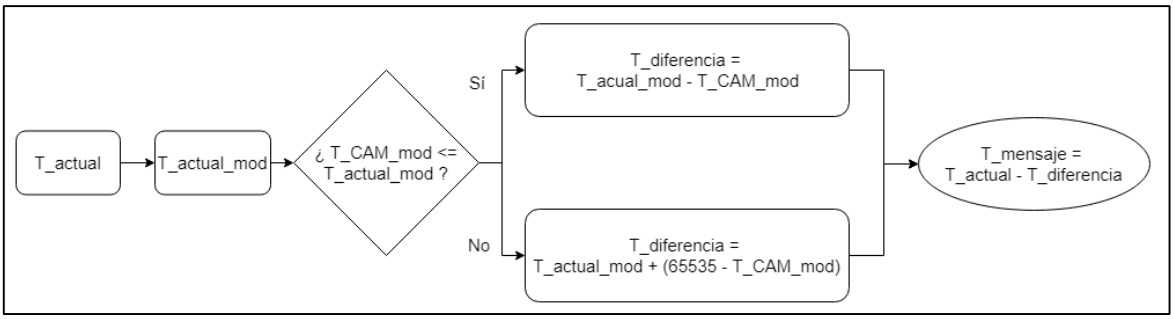

Figura 8: Esquema getRealGenerationDeltaTime

3.2.2.2. Control_entry_distance

Son las entradas que contiene control_table_distance. Se utiliza para saber las diferentes distancias a las que se encuentran el resto de vehículos a lo largo del tiempo y así saber si se están acercando o alejando de nuestra posición.Tiene:

- Atributos:

- **Id** de la estación de donde proviene el mensaje.
- **Tiempo** de creación del último mensaje recibido de ese vehículo.
- **Distancia1**: Distancia en metros más antigua que se dispone de ese vehículo.
- **Distancia2**: Segunda distancia en metros más antigua que se dispone de ese vehículo.
- **Distancia3**: Distancia en metros más nueva que se dispone de ese vehículo.
- **Distancias**: Variable de control para saber de cuantas distancias dispone la entrada.

- Métodos:

- **Getters**: Id, tiempo.
- **getDistance**(): Devuelve la última distancia que se tiene de esa entrada.
- **Print**(): Método que muestra información de la entrada con formato “id: __ || time: _____ || distances: _ || distance1: _____ || distance2: _____ || distance3: _____ “
- **setData** (tiempo, latitud, longitud, dirección, velocidad): setter de cualquier dato de la entrada.
- **newDistance**(distancia, tiempo): Este es el método al que se recurre cada vez que llega un CAM de un vecino que ya existe en la tabla de distancias. Actualiza las tres distancias de la entrada de manera adecuada y devuelve un booleano indicando si el vehículo se está acercando o alejando a partir de las distancias que dispone.

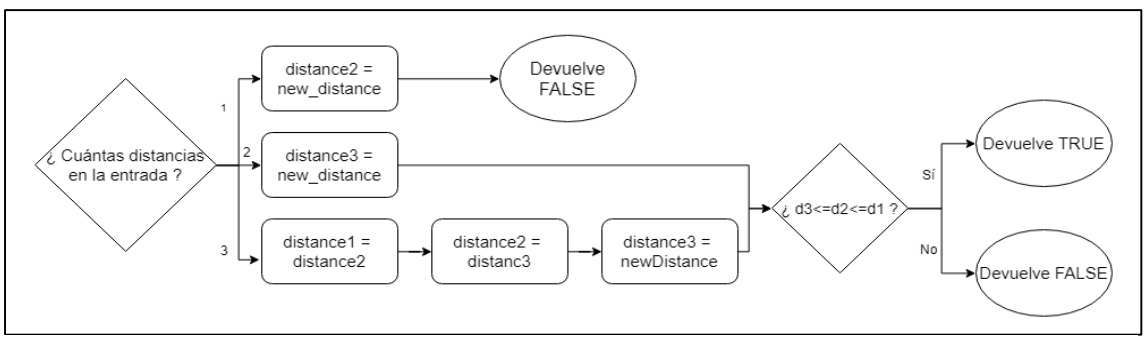

Figura 9: Esquema newDistance

3.2.2.3. Control_entry_intersection

Son las entradas que contiene control_table_intersection. Se utiliza para guardar entradas actualizadas que contienen posibles colisiones. Tiene:

- Atributos:

- **Id** de la estación de donde proviene el mensaje.
- **Tiempo** de colisión: Tiempo en milisegundos que se producirá lo colisión.
- **Latitud** de la posición donde sucederá la colisión en formato decimal.
- **Longitud** de la posición donde sucederá la colisión en formato decimal.

- Métodos:

- **Getters**: Id, tiempo, latitud, longitud.
- **Print**(): Método que muestra información de la entrada con formato "id: __ || time intersection: ____ || latitude intersection: ____ || longitude intersection: _____ ".
- **setIntersectionData** (tiempo, latitud, longitud): Setter para actualizar valores de la entrada desde la tabla de intersecciones.

3.2.3. Control_tables

Las tablas son necesarias en el programa para guardar información acerca de los vehículos del alrededor, a que distancia están y las posibles colisiones. Para el correcto funcionamiento del programa, la única tabla que ha de existir obligatoriamente es la distancias. La de vecinos e intersecciones son opcionales. Como es lógico, cada tabla guarda entradas de su tipo: control_entry_neighbour, control_entry_distance o control_entry_intersection. Aunque guarden entradas diferentes, tienen métodos muy parecidos que se explican a continuación. Después en cada tabla se explicarán las peculiaridades de cada una.

- **Print**(): Escribe por terminal el tipo de tabla que es ("neighbours table: ", "distances table: " o "intersections table: " y a continuación información de cada entrada que contiene.
- **addEntry**(control_entry): Añade al inicio de la tabla la entrada que se le pasa.
- **deleteEntry**(id): Busca en la tabla la entrada con el id pasado y lo borra.
- **checkAndAddEntry**(control_entry): Esta función difiere un poco en cada tipo de tabla. Pero, en general, comprueba si el id de la entrada que se le pasa para añadir existe en la tabla. Si es así, se modifican los datos de la entrada. Si no existía, se añade la entrada a la tabla. Más adelante, en cada tabla, se explica que hace en cada tabla en particular.
- **sortTime**(): Ordena las entradas de la tabla en función de un tiempo de menor a mayor. Dependiendo de la tabla se ordena en función de un tiempo diferente.
- **deleteByTime**(timeValue): Se ordenan las entradas por tiempo y a continuación se borran todas aquellas que tengan un valor de tiempo más pequeño que timeValue, un valor que se indica para cada tabla al iniciar el programa. Como el método anterior, este también difiere un poco en cada tabla, más adelante se explica en detalle.
- **clearTable**(): Vacía la tabla entera de entradas.

3.2.3.1. Control_table_neighbour

Esta tabla tan solo se construye cuando NEIGHBOURS_HEADING_AVAILABLE está setteada a false. Es una tabla que guarda control_entry_neighbour y se utiliza para conseguir la dirección de los vecinos cuando los mensajes CAM que recibe el programa no lo aportan correctamente.

- **checkAndAddEntry**(control_entry_neighbour): En esta tabla, cuando se detecta que hay una entrada con el mismo id se realizan tres cosas.
    - Primero calcula la dirección entre el punto que se tiene guardado en la entrada y el punto de la nueva entrada que se quiere añadir mediante el método headingBetweenTwoPoints de la clase control_entry_neighbour.
    - Después, modifica el valor de la dirección en la entrada de la tabla mediante el método setData de control_entry_neighbour.
    - Finalmente, cambia el valor de la dirección en la propia entrada que se quedará el programa para continuar el ciclo con la dirección adecuada.

  Si la entrada no existe en la tabla quiere decir que es la primera vez que se recibe un mensaje de ese vehículo o que el anterior hacía tiempo que se había recibido. Por lo tanto, como todavía no se puede saber si se está acercando, se añade la entrada a la tabla y se espera a recibir otro mensaje del mismo vecino para, entonces sí, calcular la dirección que tiene ese vecino.

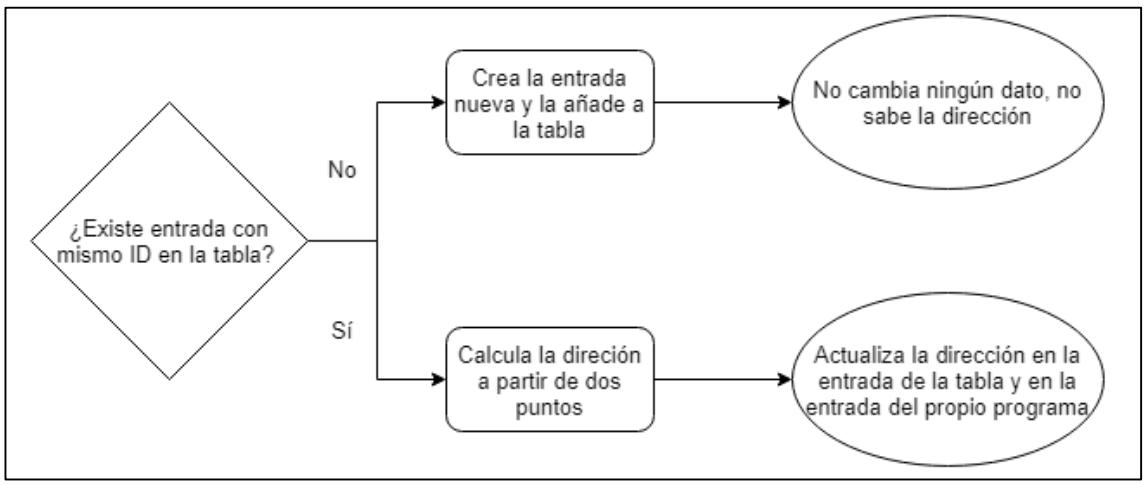

Figura 10: Esquema checkAndAddEntry (neighbour)

- **sortTime**(): Ordena las entradas en función del tiempo que se envió el CAM.
- **deleteByTime**(timeValue): Elimina todas las entradas de la tabla que hace más de TIME_LIMIT_NEIGHBOUR_TABLE milisegundos que enviaron el mensaje y no se ha actualizado. De esta manera se mantienen en la tabla vecinos que aún están cerca y, por lo tanto, han actualizado el tiempo de la entrada.

3.2.3.2. Control_table_distance

Esta tabla se construye siempre y contiene control_entry_distance. La función básica de esta tabla es proporcionarle al programa si el vecino del CAM que recibe se está acercando o alejando de su vehículo. Para hacerlo, tiene los métodos comunes explicados anteriormente y uno específico. A continuación, se explican los comunes con su peculiaridad y el específico:

- **checkAndAddEntry**(id, distancia, tiempo): Este método busca en su tabla si ya tiene guardada una entrada con el ID de la entrada que le pasan. En caso negativo, añade la entrada a la tabla y devuelve falso porque es la primera referencia que tiene de ese vecino y no puede saber si se está acercando o alejando aún. En caso de tenerlo en la tabla, recurre al método newDistance(distancia, tiempo) de control_entry_distance con el ID dado y devuelve el booleano que le da esa función en referencia a si se está acercando o alejando ese vecino.

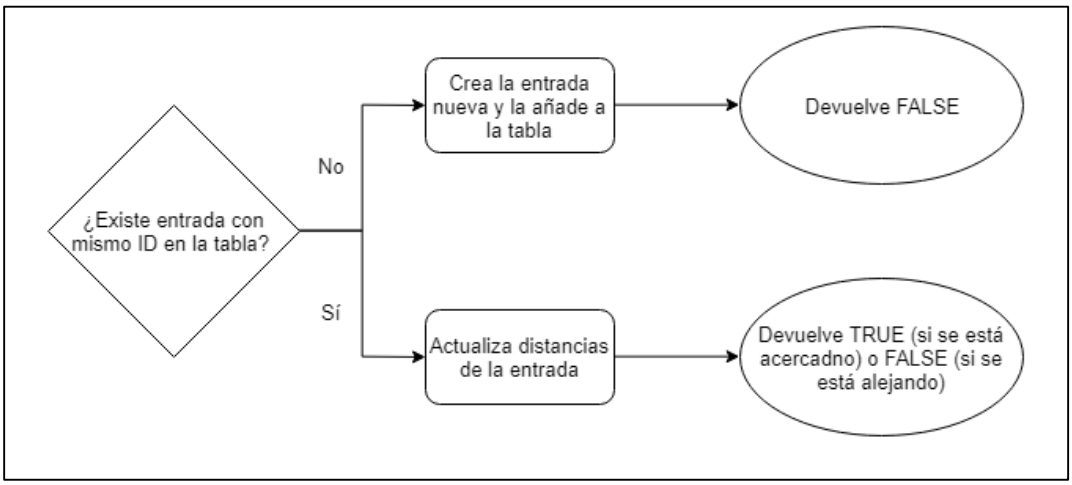

Figura 11: Esquema checkAndAddEntry (distance)

- **sortTime**(): Ordena las entradas en función del tiempo que se envió el CAM.
- **deleteByTime**(timeValue): Elimina todas las entradas de la tabla que hace más de TIME_LIMIT_DISTANCE_TABLE milisegundos que enviaron el mensaje y no se ha actualizado. De esta manera se mantienen en la tabla distancias de vecinos que aún están cerca y, por lo tanto, han actualizado el tiempo de la entrada.
- **getDistanceEntry**(id): este método busca en la tabla la entrada con el id dado y llama al método getDistance del correspondiente control_entry_distance que devuelve la última distancia de ese vecino.

3.2.3.3. Control_table_intersection

Esta tabla se construye solo cuando la variable booleana SAVE_INTERSECTION_TABLE está setteada a cierto y contiene control_entry_intersection. La función de esta tabla es mantener un registro de las posibles colisiones. Para hacerlo, tiene los métodos comunes explicados anteriormente. A continuación, se explican los comunes con su peculiaridad:

- **checkAndAddEntry**(control_entry_intersection): Este método busca en su tabla si ya tiene guardada una entrada con el ID de la entrada que le pasan. En caso negativo, añade la entrada a la tabla. En caso de tenerlo en la tabla, recurre al método setIntersectionData(tiempo, latitud, longitud) de control_entry_intersection con el ID dado para así actualizar los datos de esa entrada.
- **sortTime**(): Ordena las entradas en función del tiempo que se producirá la colisión.
- **deleteByTime**(): Elimina todas las entradas de la tabla con el valor de tiempo anterior al actual porque son colisiones que no se han producido.

3.2.4. Auxiliar GPS

En esta clase se definen un conjunto de métodos para calcular distancias, direcciones o puntos geográficos según una dirección.

Para ello, contiene 3 estructuras que simulan diferentes puntos con atributos diferentes:

- **simple_point**: Estructura más simple, solo define posición geográfica. Tiene:
  - latitude.
  - Longitude.
- **bearing_point**: Estructura que contiene posición geográfica y dirección del vehículo. Tiene:
  - latitude.
  - Longitude.
  - Bearing.
- **speed_point**: Estructura que contiene posición geográfica y velocidad del vehículo. Tiene:
  - latitude.
  - Longitude.
  - Speed.

También tiene dos constantes que se utilizan para diferentes operaciones:

- **PI**: Valor con más de 30 decimales de la librería GPS.
- **EARTHRADIUS:** 6378137 metros.

En cuanto a métodos, hay dos que son simples conversores de unidades que utilizan una conversión con más de 30 decimales de la librería GPS:

- double **radToDeg**(radians).
- double **degToRad**(degrees).

El resto de métodos son la parte importante de la clase:

- double **distance**(simple_point, simple_point): Devuelve la distancia en metros entre dos puntos geográficos. Para calcularla se ha utilizado la distancia de haversine adaptada al lenguaje de programación:

Haversine formula:

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$$
$$c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{(1-a)})$$
$$d = R \cdot c$$

where φ *is latitude,* λ *is longitude,* R *is earth's radius*

Figura 12: Fórmula distancia Haversine [10]

- simple_point **intersection**(bearing_point, bearing_point): Devuelve el punto de colisión partiendo desde dos puntos geográficos y si ambos vehículos mantienen la dirección constante dada en los puntos iniciales. Para calcularla se ha utilizado la siguiente fórmula adaptada al lenguaje de programación:

Formula: $\delta_{12} = 2 \cdot \text{asin}(\sqrt{(\sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2))})$ — angular dist. p1–p2

$\theta_a = \text{acos}((\sin\varphi_2 - \sin\varphi_1 \cdot \cos\delta_{12}) / (\sin\delta_{12} \cdot \cos\varphi_1))$
$\theta_b = \text{acos}((\sin\varphi_1 - \sin\varphi_2 \cdot \cos\delta_{12}) / (\sin\delta_{12} \cdot \cos\varphi_2))$ — initial / final bearings between points 1 & 2

if $\sin(\lambda_2 - \lambda_1) > 0$
  $\theta_{12} = \theta_a$
  $\theta_{21} = 2\pi - \theta_b$
else
  $\theta_{12} = 2\pi - \theta_a$
  $\theta_{21} = \theta_b$

$\alpha_1 = \theta_{13} - \theta_{12}$ — angle p2–p1–p3
$\alpha_2 = \theta_{21} - \theta_{23}$ — angle p1–p2–p3

$\alpha_3 = \text{acos}(-\cos\alpha_1 \cdot \cos\alpha_2 + \sin\alpha_1 \cdot \sin\alpha_2 \cdot \cos\delta_{12})$ — angle p1–p2–p3

$\delta_{13} = \text{atan2}(\sin\delta_{12} \cdot \sin\alpha_1 \cdot \sin\alpha_2 , \cos\alpha_2 + \cos\alpha_1 \cdot \cos\alpha_3)$ — angular dist. p1–p3

$\varphi_3 = \text{asin}(\sin\varphi_1 \cdot \cos\delta_{13} + \cos\varphi_1 \cdot \sin\delta_{13} \cdot \cos\theta_{13})$ — p3 lat

$\Delta\lambda_{13} = \text{atan2}(\sin\theta_{13} \cdot \sin\delta_{13} \cdot \cos\varphi_1 , \cos\delta_{13} - \sin\varphi_1 \cdot \sin\varphi_3)$ — long p1–p3

$\lambda_3 = \lambda_1 + \Delta\lambda_{13}$ — p3 long

where φ_1, λ_1, θ_{13} : 1st start point & (initial) bearing from 1st point towards intersection point
φ_2, λ_2, θ_{23} : 2nd start point & (initial) bearing from 2nd point towards intersection point
φ_3, λ_3 : intersection point

Figura 13: Fórmula intersección dos direcciones en un escenario esférico [10]

- simple_point **pointFollowingBearing**(bearing_point, double): Devuelve la posición geográfica final si desde la posición geográfica inicial se sigue la dirección constante dada durante los metros dados.
  Después de probar varias fórmulas, se vio que al ser distancias cortas lo mejor era calcularlo como si fuera en dos dimensiones. Además, se realizó una herramienta aparte con bibliotecas de software libre sobre mapas y los resultados eran iguales, por tanto, se dio por bueno el resultado.

$$latitude = latitudeStart + degToRad\left(\frac{distance}{110540}\right) * \cos(bearing)$$
$$longitude = longitudeStart + degToRad\left(\frac{distance}{11320 * \cos(latitudeStart)}\right) * \sin(bearing)$$

- double **timeToPoint**(speed_point, simple_point): Devuelve el tiempo en segundos que se tardaría a llegar desde el primer punto hasta el segundo si se mantiene la velocidad constante dada.
  Para hacerlo, simplemente calcula la distancia entre los dos puntos y después la divide por la velocidad del propio vehículo.
- bool **getBackPoint**(bearing_point, bearing_point): Método que a partir de dos posiciones geográficas y sus respectivas direcciones en un momento concreto devuelve que vehículo está detrás del otro. Si la función devuelve FALSE, el segundo punto se encuentra detrás; sí devuelve TRUE, el primero se encuentra detrás.

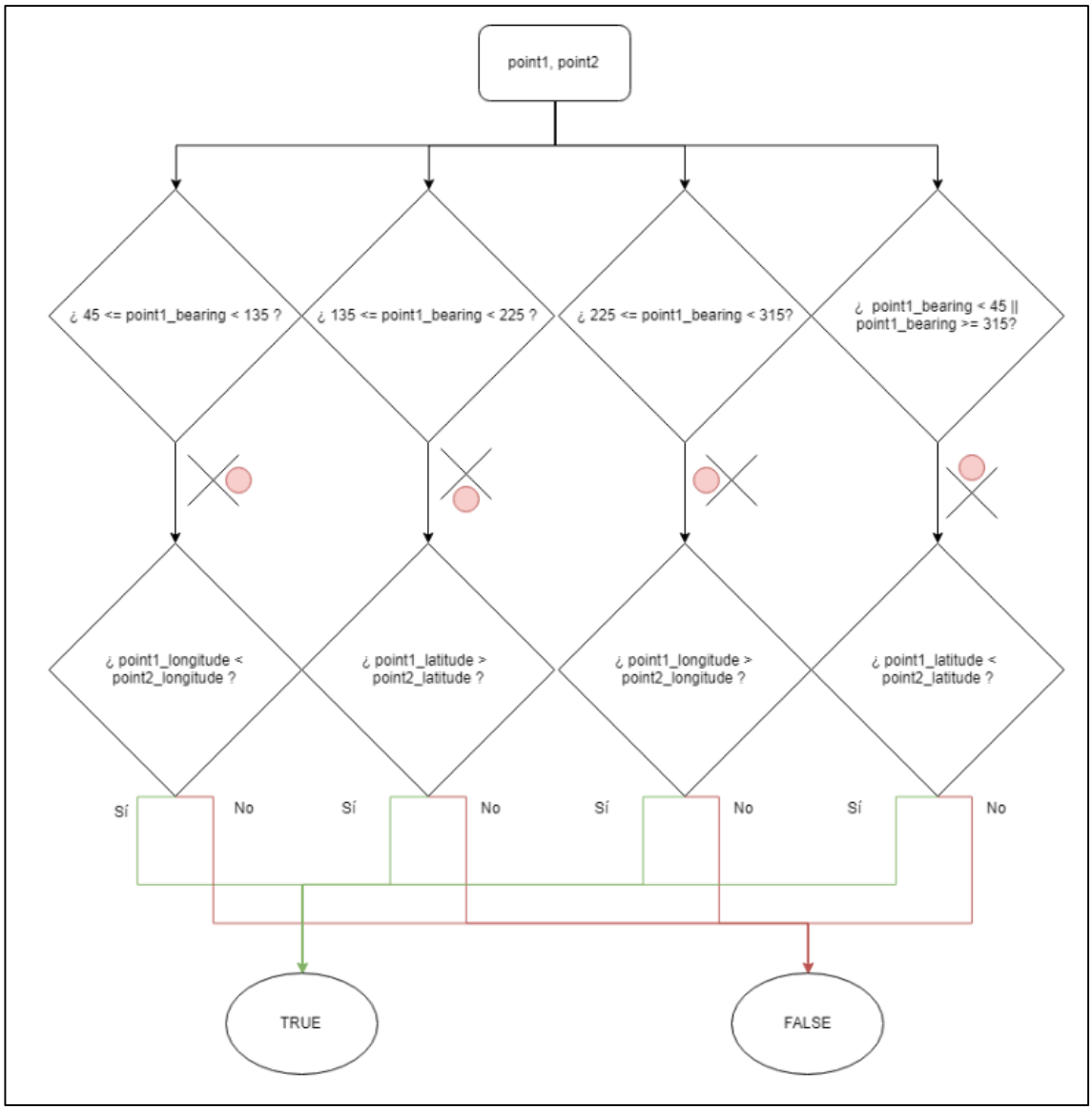

Figura 14: Esquema getBackPoint

- double **bearingBetweenPoints**(simple_point, simple_point): Devuelve la dirección en grados que une en línea recta la primera posición geográfica con la segunda. Para calcularla se ha utilizado la siguiente fórmula adaptada al lenguaje de programación:

Formula: $\theta = \operatorname{atan2}(\sin\Delta\lambda \cdot \cos\varphi_2 , \cos\varphi_1 \cdot \sin\varphi_2 - \sin\varphi_1 \cdot \cos\varphi_2 \cdot \cos\Delta\lambda)$

where φ_1,λ_1 *is the start point,* φ_2,λ_2 *the end point (*$\Delta\lambda$ *is the difference in longitude)*

Figura 15: Fórmula dirección entre dos puntos en un escenario esférico [10]

3.2.5 Controller

Es la clase principal de la aplicación, la que controla todas las operaciones. Contiene las tres tablas explicadas, los datos de posición, velocidad y dirección del propio vehículo, las variables de la aplicación que son iniciadas en su constructor y un conjunto de métodos que serán explicados a continuación.

- Atributos:

- tablas: **neighbours_**, **distances_**, **intersections_**.
- información del vehículo: **latitude_**, **longitude_**, **bearing_**, **speed_**.
- constantes numéricas: **TIME_LIMIT_NEIGHBOUR_TABLE**, **TIME_LIMIT_DISTANCE_TABLE**, **TIME_INTERVAL_INTERSECTION TIME_LIMIT_INTERSECTION_TABLE**, **BEARING_LIMIT**.
- constantes booleanas: **SAVE_INTERSECTION_TABLE**, **NEIGHBOURS_HEADING_AVAILABLE**, **OWN_HEADING_AVAILABLE**.

- Estructura:
  control_intersection:{
    - bool intersection
    - control_entry_intersection

  }

  Esta estructura se utiliza en el controller para saber si hay peligro de colisión o no después de realizar todas las operaciones. Si el booleano es FALSE, significa que no hay peligro de colisión y, entonces, control_entry_intersection tendrá seteado a 0 todos sus atributos. Si, por el contrario, el booleano es TRUE, significa que hay que avisar de la posible colisión y todos los datos (id del vehículo, tiempo que se producirá la colisión, posición geográfica de la colisión) estarán seteados en la control_entry_intersection.

- Métodos:

- void **updateValues**(): Función que actualiza los cuatro atributos sobre posición del vehículo. Latitud, longitud y velocidad los coge directamente del GPS. Pero la dirección depende de la variable OWN_HEADING_AVAILABLE: si es TRUE, coge la dirección del GPS también; si es FALSE, recurre a la función calculateBearing(double, double) para conseguirla.
- double **calculateBearing**(double newLatitude, double newLongitude): Calcula la dirección del vehículo cogiendo como punto inicial los valores de latitud y longitud que tenía guardados y como punto destino la latitud y longitud que se le pasan como parámetro.
- void **newNeighbour**(control_entry_neighbour): Método principal del programa. Al llegar un mensaje CAM es el método que se llama y que va realizando todas las operaciones explicadas en el flujo.
- bool **checkDistance**(control_entry_neighbour): Este método recibe un control_entry_neighbour y ha de devolver un booleano indicando si el vecino se está acercando o alejando. Para hacerlo, crea dos simple_point, uno con los datos propios y otro a partir de la entrada que le han pasado. Entonces ya puede calcular la distancia que hay en ese momento entre ambos vehículos. Una vez tiene la distancia calculada, recurre a la función checkAndAddEntry de la tabla de distancias que ya hace las operaciones necesarias para saber si se está acercando o alejando.

- control_intersection **checkTimeIntersectionPoint**(control_entry_neighbour): Este método es llamado una vez sabemos que el vehículo se está acercando. Entonces, dependiendo de cómo se acerca hay que dirigir el cálculo del punto de colisión de distinta manera. En la figura 26 está el esquema del cálculo.

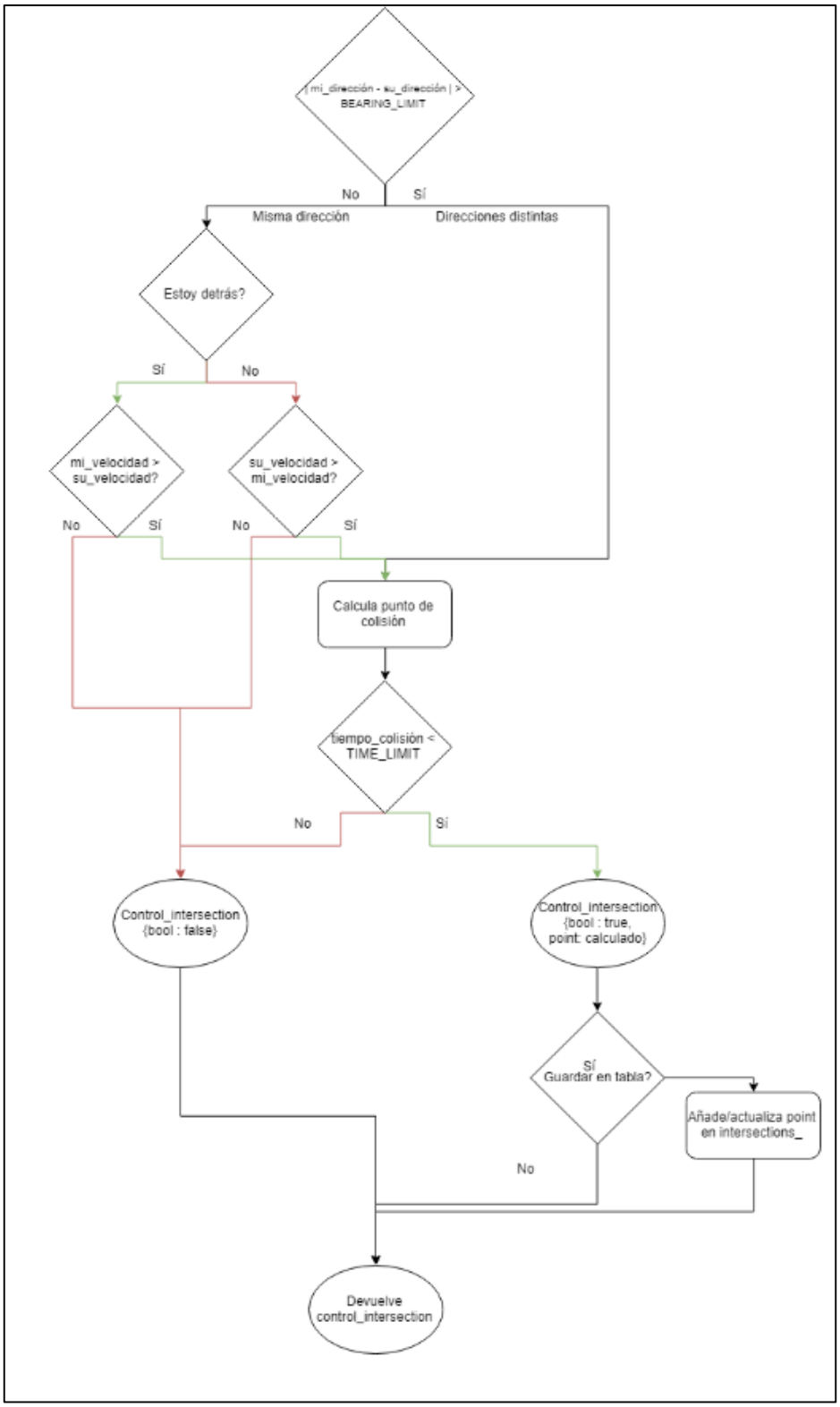

Figura 16: Esquema checkTimeIntersectionPoint

- métodos de gestión de la tabla neighbours_:

    - void **updateNeighbourTable**().
    - void **clearNeighbourTable**().
    - void **printNeighbourTable**().

- métodos de gestión de la tabla distances_:

    - void **updateDistanceTable**().
    - void **clearDistanceTable**().
    - void **printDistanceTable**().

- métodos de gestión de la tabla intersections_:

    - void **updateIntersectionTable**().
    - void **clearIntersectionTable**().
    - void **printIntersectionTable**().

3.2.5. Variables de la aplicación

La aplicación presentada se rige en función de unas variables configurables por código en el constructor. Hay algunas que podrían irse modificando durante la ejecución de la aplicación, pero en esta versión de la aplicación no está ha hecho así. Hay dos tipos de variables configurables: numéricas y booleanas:

- Númericas:

    - **TIME_LIMIT_NEIGHBOUR_TABLE** (milisegundos): Variable que rige el tiempo máximo sin un nuevo mensaje de los vecinos que hay guardados en la tabla de vecinos.
    - **TIME_LIMIT_DISTANCE_TABLE** (milisegundos): Variable que rige el tiempo máximo sin un nuevo mensaje de los vecinos que hay guardados en la tabla de distancias.
    - **TIME_LIMIT_INTERSECTION_TABLE** (segundos): Variable que rige el tiempo máximo en el que se producirá una colisión para mostrar el choque o no.
    - **TIME_INTERVAL_INTERSECTION** (segundos): Variable que rige el tiempo de diferencia que ha de haber entre la llegada de dos vehículos a un punto para considerar colisión o no.
    - **BEARING_LIMIT** (grados): Variable que marca el límite para considerar que dos vehículos circulan en la misma dirección.
- Booleanas:

    - **SAVE_INTERSECTION_TABLE**: Variable que decide si guardar las colisiones en una tabla o no.
    - **NEIGHBOURS_HEADING_AVAILABLE**: Variable que decide si el dato de la dirección que recibe del mensaje CAM es aceptado o no. Esta variable influye en la creación o no de la tabla de vecinos. Si es cierta, como acepta la dirección del mensaje, no hace falta mantener la tabla; si es falsa, como no acepta la dirección de los mensajes, necesita la tabla para calcularla a partir de las posiciones.

- **OWN_HEADING_AVAILABLE**: Variable que decide si aceptar o no la dirección que aporta el propio GPS. Si es cierto, se acepta el dato; si es falso, se calcula la dirección a partir de la posición anterior guardada.

3.3. Modificación Vanetza heading/speed

Durante la realización del programa, para comprobar si la aplicación funcionaba correctamente se probaba todo en local. Al tenerla ya acabada, se comenzó a probar la aplicación conectada a GPS reales. Entonces se vio que el proyecto Vanetza de Github creaba los mensajes CAM poniendo la dirección y la velocidad del vehículo siempre a 0, no tenía en cuenta el valor que le proporcionaba el GPS.

Así que para que el programa funcionara con todos los datos correctos, se hizo una modificación en el código de Vanetza.

```
bvc.heading.headingValue = 0
bvc.heading.headingConfidence = HeadingConfidence_equalOrWithinOneDegree;

bvc.speed.speedValue = 0
bvc.speed.speedConfidence = SpeedConfidence_equalOrWithinOneCentimeterPerSec;
```

Figura 17: Asignación dirección y velocidad en CAM original Vanetza

El código original va leyendo datos del GPS y los guarda en la estructura de la figura 18. Se puede ver que la velocidad y dirección las guarda en una clase ConfidentQuantity. Esto es porque estos dos valores además del valor han de decir el nivel de confianza con el cual está tomado el valor.

```
struct PositionFix
{
    Clock::time_point timestamp;
    units::GeoAngle latitude;
    units::GeoAngle longitude;
    PositionConfidence confidence;
    ConfidentQuantity<units::TrueNorth> course;
    ConfidentQuantity<units::Velocity> speed;
};
```

Figura 18: EstructuraPositionFix Vanetza

Después de muchas pruebas, no se encontró la manera de extraer el valor de esa clase. Así que se optó por crear una estructura similar pero que no tuviera en cuenta en nivel de confianza. De esta manera, cada vez que se guardará una posición se rellenarían dos estructuras. No se podía guardar solo en la nueva porque realmente hay que comprobar el nivel de confianza, y para eso ya existe la estructura creada y un conjunto de métodos que lo comprueban.

```
struct PositionFixComplete
{
    Clock::time_point timestamp;
    units::GeoAngle latitude;
    units::GeoAngle longitude;
    PositionConfidence confidence;
    units::TrueNorth  course;
    units::Velocity speed;
};
```

Figura 19: Estructura nueva PositionFix Vanetza

```
bvc.heading.headingValue = position_fix_complete.course.value();
bvc.heading.headingConfidence = HeadingConfidence_equalOrWithinOneDegree;

bvc.speed.speedValue = position_fix_complete.speed.value() * 100;
bvc.speed.speedConfidence = SpeedConfidence_equalOrWithinOneCentimeterPerSec;
```

Figura 20: Asignación dirección y velocidad en CAM modificada Vanetza

4. Resultados

4.1. Página web comprobaciones con JavaScript

Para comprobar si los datos que proporcionaba el programa eran correctos, se creó una página web simple con Javascript utilizando la librería Turf JS para los cálculos I implementando OpenStreetMaps para la representación gráfica. De esta manera se tenía otra fuente de resultados independiente para comprobar los resultados que devolvían las funciones del auxiliar_gps. Antes de crear esta herramienta se comprobaba con la página web nmeagen.org. Pero los resultados no eran exactamente iguales, y al crear la página web se vio que los datos que proporcionaba nmeagen.org acerca de la dirección no eran del todo correctos. La página web creada comprueba tres funciones de auxiliar_gps:

- **Distancia entre dos puntos**

Esta opción necesita dos latitudes y dos longitudes y te devuelve la distancia entre ambos puntos en metros. En la figura 21 se puede ver un ejemplo.

Figura 21: Javascript - Distancia

- **Intersección**

Esta opción calcula la intersección entre dos puntos con una dirección determinada. En la figura 22 se quiere calcular la intersección de un vehículo representado por el punto superior izquierdo y otro simulado por el punto superior derecho. Según la página web utilizada en un principio colisionarían en el punto inferior con las coordenadas que se ven en el cuadro inferior.

Pero al introducir los datos en la página web creada con javascript el resultado se puede ver en la figura 23.

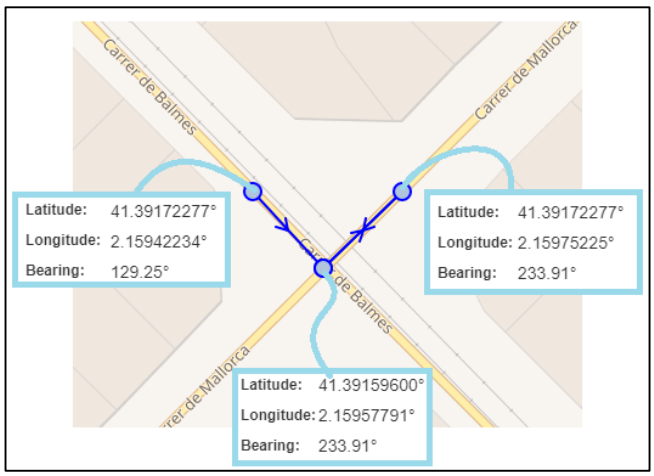

Figura 22: Cálculo intersección página web externa

Figura 23: Javascript - Intersección

Se puede ver que el punto resultante es diferente. Además, se observa que la dirección que ofrece la página web no es correcta porque al poner las líneas rectas que la siguen, no continúan el curso de la calle, como si parece en la página web nmeagen.org.

Utilizando la opción explicada en el punto anterior se observa que la diferencia entre los puntos dados en la página web externa i la creada es de casi 3.5 metros. La función creada en auxiliar_Gps da el mismo resultado que la página web con javascript propia.

- **Punto siguiendo una dirección**

Esta opción se le proporciona un punto con su latitud y longitud y además una dirección y una distancia en kilómetros. La herramienta devuelve el punto geográfico con su latitud y longitud.

Figura 24: Javascipt - Punto siguiendo dirección

Figura 25: Javascipt - comprobación punto siguiendo dirección

En la figura 25 se demuestra que la distancia es correcta utilizando la primera opción explicada de la página web.

4.2. Rendimiento en tiempo de la aplicación

A continuación, se muestran los resultados que se han obtenido acerca del tiempo necesario para procesar un mensaje en dos situaciones. Estos tiempos son la media de 100 mensajes en cada situación y con las variables adaptando un valor adecuado al mundo real (2 s en las tablas y sin guardar tabla de intersecciones). Con este tiempo se podría saber el número máximo de vehículos que puede procesar sabiendo que cada vehículo envía 10 mensajes por segundo. Aunque a este tiempo hay que añadirle algo más porque solo incluye el tiempo de proceso del programa, no del receptor entero (parser de la figura 5 excluido).

Time1 indica el tiempo que tardaría cuando el programa no tiene ningún dato acerca de ese vehículo. Time2 indica lo que tardaría cuando se tiene un dato en la tabla de distancias. Y time 3 cuando se tiene 2 datos en la tabla distancias y, por lo tanto, ya se puede calcular intersección si el vehículo se acerca. n indica el número de entradas que contiene la tabla de distancias.

- **Situación 1: Vehículo acercándose**

n	time1(s)	time2(s)	time3(s)
5	3,03E-02	2,87E-02	3,83E-02
15	5,48E-02	5,19E-02	6,15E-02
25	7,88E-02	7,46E-02	8,42E-02
35	1,08E-04	1,02E-04	1,12E-04
45	1,29E-04	1,22E-04	1,32E-04
55	1,55E-04	1,46E-04	1,56E-04
65	1,92E-04	1,82E-04	1,92E-04
75	2,13E-04	2,02E-04	2,12E-04
85	2,38E-04	2,25E-04	2,35E-04
95	2,62E-04	2,48E-04	2,58E-04

Tabla 2: Rendimiento tiempo vehículo acercándose

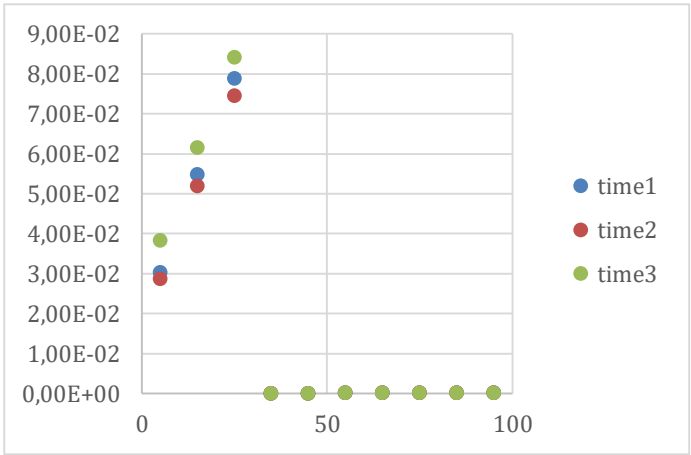

Figura 24: Rendimiento tiempo vehículo acercándose

Se puede ver que tarda más en procesar un mensaje cuando ya tiene dos distancias guardadas. También se ve que tarda más cuando no tiene distancias que cuando tiene 1, esto es debido a que cuando no tiene nada ha de crear la entrada y cuando si la tiene solo ha de modificar dos datos. Por último, también se aprecia que hasta los 25 mensajes va incrementando el tiempo de proceso a medida que tiene más distancias guardadas, pero alrededor de 35 hace un salto considerable a tardar menos tiempo para volver a incrementar el tiempo lentamente a medida que tiene más distancias guardadas.

- **Situación 2: Vehículo alejándose**

n	time1(s)	time2(s)	time3(s)
5	3,03E-02	2,94E-02	2,88E-02
15	5,47E-02	5,26E-02	5,19E-02
25	7,89E-02	7,53E-05	7,47E-02
35	1,08E-04	1,03E-04	1,02E-04
45	1,29E-04	1,23E-04	1,22E-04
55	1,54E-04	1,47E-04	1,46E-04
65	1,92E-04	1,82E-04	1,82E-04
75	2,13E-04	2,03E-04	2,02E-04
85	2,37E-04	2,26E-04	2,25E-04
95	2,62E-04	2,48E-04	2,48E-04

Tabla 3: Rendimiento tiempo vehículo alejándose

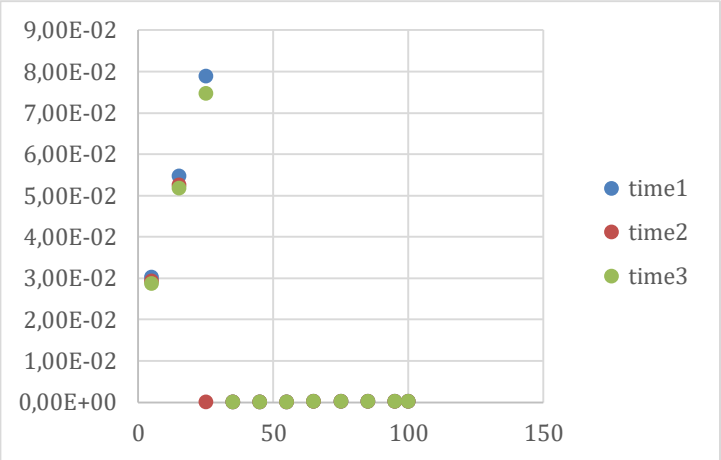

Figura 25: Rendimiento tiempo vehículo alejándose

En este se observa que el comportamiento respecto el número de vehículos guardados es el mismo. Pero la diferencia de tiempos según el número de distancias guardadas es diferente. Aquí siempre tarda más la primera vez respecto el resto.

4.3. Aplicación con datos manuales

Al probar la aplicación pasándole el control_entry_neighbour totalmente manual (como los casos en que se ha probado para hacer los gráficos del tiempo de uso) el programa funciona correctamente con todas las opciones posibles. Los casos simulados, como puede verse en las figuras, están probados utilizando las interfaces localhost y simulando el gps con un archivo de texto que simulaba las trayectorias deseadas.

En la figura 28 se puede observar el aviso de colisión en caso de tener dos vehículos que circulan en dos trayectorias perpendiculares.

```
sergi@sergi-Aspire-A315-51:~/Escritorio/camReceiver/build$ sudo bin/socktap-cam -i lo --gpsd-host localhost --gpsd-port 5000
CAM receiver application running
Reading options...
Initialising CAM receiver...
Starting runtime at 2019-May-29 17:51:55.322489
CHOQUE:id: 1  time intersection: 1.797308785  latitude intersection41.39079695  longitude intersection2.158467722
```

Figura 26: Demostración aplicación datos manuales (1/2)

En la figura 29 se puede observar el aviso de colisión cuando un vehículo circula detrás de otro más rápido y puede producirse una colisión frontal.

```
sergi@sergi-Aspire-A315-51:~/Escritorio/camReceiver/build$ sudo bin/socktap-cam -i lo --gpsd-host localhost --gpsd-port 5000
CAM receiver application running
Reading options...
Initialising CAM receiver...
Starting runtime at 2019-May-29 17:55:22.050124
CHOQUE:id: 1  time intersection: 1.318098314  latitude intersection: 41.39080509  longitude intersection: 2.158482125
```

Figura 27: Demostración aplicación datos manuales (2/2)

4.4. Aplicación con datos reales

También se realizaron pruebas simulando un caso real (representado en la figura 5): 2 Raspberry Pis y dos GPS independientes conectados a la misma red inalámbrica. Para probar se hizo andando, por lo tanto, la velocidad es mucho menor que la de un vehículo y también se subió el tiempo de colisión a 5 segundos porque a la velocidad que se probaba el choque era más tarde que si fuera en un vehículo a motor.

Las pruebas se hicieron simulando las 4 direcciones que se muestran en la figura 30.

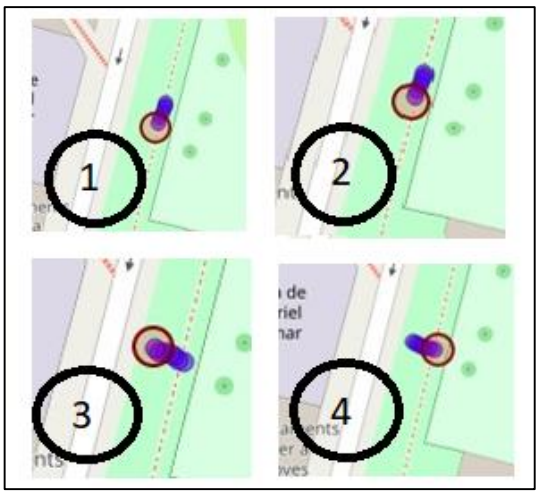

Figura 28: Trayectorias probadas

El programa notifica la colisión, pero no en el punto correcto combinando cualquiera de las 4 trayectorias.

```
sergi@sergi-Aspire-A315-51:~/Escritorio/camReceiver/build$ sudo bin/socktap-cam -i wlp3s0 --gpsd-host localhost
CAM receiver application running
Reading options...
Initialising CAM receiver...
Starting runtime at 2019-Jun-08 14:47:32.694776
CHOQUE:id: 1  time intersection: 3.67844  latitude intersection41.4374  longitude intersection2.18286
CHOQUE:id: 1  time intersection: 1.14428  latitude intersection41.4375  longitude intersection2.18286
CHOQUE:id: 1  time intersection: 2.89603  latitude intersection41.4374  longitude intersection2.18287
```

Figura 29: Demostración aplicación datos reales (1/2)

Además, también se probó la trayectoria 1 a diferentes velocidades para comprobar si notificaba la colisión frontal. En este escenario nunca notificó posible choque. Para descubrir por qué no lo hacía, se mostraron valores de longitud y latitud y se descubrió el porqué. El programa no recibe la posición con los decimales correctos y calcula mal la dirección.

```
lat: 41.4375 - lon:2.18288  heading: 2.47059
lat: 41.4375 - lon:2.18287 heading: 200.917
lat: 41.4375 - lon:2.18286 heading: 193.737
lat: 41.4375 - lon:2.18286 heading: 0
lat: 41.4375 - lon:2.18289 heading: 18.027
lat: 41.4375 - lon:2.18289 heading: 0
```

Figura 30: Demostración aplicación datos reales (2/2)

En caso de poner dirección a true también falla porque la dirección que proporciona el GPS nunca era fija, cuando realmente sí lo era.

5. Presupuesto

Para el cálculo de los costes se ha tenido en cuenta los dispositivos necesarios para realizar las pruebas, el material para realizar el programa y las horas trabajadas.

A continuación, una tabla con los costes amortizados del material:

	Precio	Amortizado
Ordenador	500,00 €	33,33 €
Mobiliario	200,00 €	6,67 €
Material oficina	10,00 €	10,00 €
		50,00 €

Tabla 4: Amortizaciones

El tiempo de amortización es de 4 meses. El ordenador se tiene en cuenta que se puede utilizar durante 5 años y el mobiliario durante 10 años.

La tabla con resumen de todos los costes:

	Precio	Cantidad	Total
Raspberry Pi	35,00 €	2,00 €	70,00 €
Módulo GPS	11,38 €	2,00 €	22,76 €
Horas trabajo	10,00 €	540,00 €	5.400,00 €
Material			50,00 €
			5.542,76 €

Tabla 5: Presupuesto total

6. Conclusiones y mejoras futuras

A partir de los resultados con la página web se puede ver que los resultados que aporta el programa acerca de distancias, direcciones y puntos geográficos son correctos porque una librería externa aporta los mismos datos.

En el apartado del tiempo de proceso de los mensajes se puede ver que para pocos vehículos alrededor la aplicación funcionaría mal porque solo sería capaz de procesar 2/3 mensajes cada 100 ms. Un valor totalmente insuficiente. Sin embargo, a partir de 35 vehículos cerca, el tiempo de proceso ya es suficiente.

Los resultados también aportan que el programa funciona correctamente si los datos que se le facilitan tienen la suficiente precisión. Sin embargo, al probarlo en el mundo real con datos aportados por GPS externos, se ve que el programa necesita mejorar sus prestaciones.

Por lo tanto, se tiene una primera versión de una aplicación que funciona correctamente en algunos escenarios. A continuación, se muestran las mejoras necesarias y otras más secundarias que necesitaría la aplicación para funcionar en el mundo real:

- Mejorar el rendimiento de la aplicación en el tiempo de proceso con pocos mensajes. Investigar porque las listas en C++ (como están programas las tablas) necesitan más tiempo con pocas entradas, o encontrar una alternativa económica a la Raspberry Pi que procese más rápido los mensajes.
- La aplicación necesita más precisión geográfica para calcular correctamente los puntos de colisión y para saber si ha de calcular la colisión como si dos vehículos llevan la misma dirección (colisión frontal) o con direcciones distintas.
- La aplicación actual funciona mal si circulan dos coches en paralelo por carriles distintos. Ahora mismo no hay ninguna comprobación para evitar eso y la aplicación detectaría que circulan en la misma dirección y calcularía la colisión como si uno estuviera delante del otro. Para solucionarlo se debe crear un algoritmo para detectar que circulan en paralelo y no uno detrás de otro.
- Otra mejora aplicable podría ser adaptar el funcionamiento al tipo de vía. Porque si se circula por una autopista, se podría saber por la velocidad que se circula o por la señal GPS comprobando algún servicio de mapas, está claro que no se puede colisionar con coches que circulan en dirección opuesta. Por lo tanto, se podría mejorar el rendimiento de la aplicación evitando calcular colisiones con coches que no circulen en la misma dirección. Con la misma lógica, se podría poner una distancia mínima para calcular colisiones si la velocidad del vehículo es baja porque ya sabes de antemano que el tiempo de la colisión va a ser muy alto.
- El programa funciona bien y no hay errores en programación. Pero como todos los proyectos que incluyen programación, son siempre mejorables. A parte del tema de PositionFixComplete se puede intentar mejorar el rendimiento en tiempo de la aplicación con más conocimientos de C++. También se puede incorporar interfaces en el caso de las tablas y entradas.
- En el proyecto estaba pensado para mostrar la posible colisión en una aplicación externa que funcionara en un tablet conectada a la Raspberry Pi. Pero al final no se pudo hacer está aplicación final. Sería una mejora considerable tener una aplicación que te mostrará de donde viene el peligro y consejos para evitarla.

Bibliografía

ETSI Standards:

[1] ETSI EN 302 637-2 v1.3.1 "Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specifications of Cooperative Awareness Basic Service", September 2019.

[2] ETSI TS 102 894-2 v1.2.1 "Intelligent Transport Systems (ITS); Users and applications requirements; Part 2: Applications and facilities layer common data dictionary", September 2019.

Informes proyecto "V2X-Arch by Cellnex":

[3] i2Cat, " Evaluación experimental de arquitecturas y tecnologías de comunicaciones V2X en entornos de Internet del Futuro WP2-E2.1", V2X-Arch by Cellnex, Diciembre 2015.

[4] i2Cat, "Evaluación experimental de arquitecturas y tecnologías de comunicaciones V2X en entornos de Internet del Futuro WP2-E2.2", V2X-Arch by Cellnex, Diciembre 2015.

Transparencias académicas:

[5] Jordi Casademont "The Connected Vehicle (communications V2X to provide C-ITS)", 2017-2018

Conferencia:

[6] Joost Vantomme (Smart mobility director en ACEA) "European momentum for ICV", Mayo 2019: http://www.oica.net/wp-content/uploads/European-momentum-ACEA-Joost-Vantomme.pdf

Repositorio Vanetza;

[7] https://github.com/riebl/vanetza

Trabajo final de máster

[8] Ugnaya Mendoza, Charmae Franchesca, "Implementation and experimental evaluation of Cooperative Awareness Basic Service for V2X Communications", UPC Commons, Mayo 2019.

Páginas web:

[9] https://nmeagen.org/ : coordenadas y generador archivos GPS.

[10] https://www.movable-type.co.uk/scripts/latlong.html :fórmulas matemáticas.

[11] https://rl.se/gprmc : decodificador formato GPRMC y GPGCA.

[12] http://aprs.gids.nl/nmea/ : formatos GPS.

Glosario

ASN.1: Abstract Syntax Notation One

CAM: Cooperative Awareness Message

CEN: Comité Europeo de Normalización

C-ITS: Cooperative Intelligent Tansport System

ETSI: European Telecommunications Standards Institute

IEEE: Institute of Electrical and Electronics Engineers

ISO: International Organization for Standardization

ITS: Intelligent Transport System

JSAE: Society of Automotive Engineers of Japan

OBU: On Board Unit

PDU: Protocol Data Unit

RSU: Road Site Unit

TCP: Transmission Control Protocol

WGS 84: World Geodetic System 1984

ANEXO 1: Arquitectura C-ITS de la ETSI

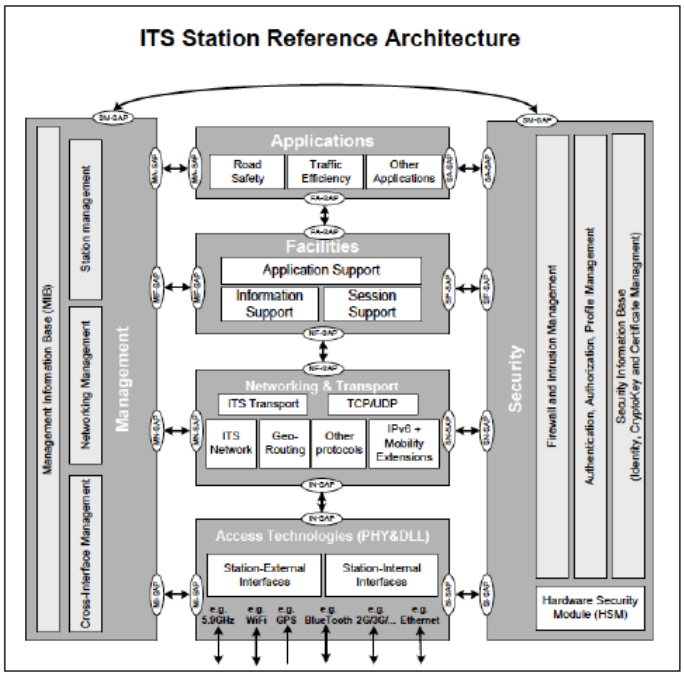

Figura 31: Arquitectura en capas de un sistema C-ITS [4]

Las dos capas verticales son:

Management/Gestión: encargada del intercambio de información entre las capas horizontales.

Security/Seguridad: proporciona seguridad y privacidad a los usuarios que utilizan estas tecnologías en todas las capas.

Las cuatro capas horizontales son:

Capa aplicaciones:

Esta capa incluye los servicios destinados al usuario final. Actualmente hay dos aplicaciones estandarizadas:

- Data Excange (DATEX II): estándar para intercambio de datos de tráfico e información de viajes entre centros de gestión de tráfico e infraestructuras de control de tráfico como los paneles luminosos.
- Transport Protocol Expert Group (TPEG): ofrece un método para transmitir información de tráfico y de viaje desde un proveedor de servicio a múltiples nodos independientemente del tipo de dispositivo cliente, localización o tecnología de acceso.

Capa facilidades:

Esta capa incluye funciones de soporte a las aplicaciones que involucran acceso a datos de otros sistemas (sensores del coche, de la carretera, información de mapas…). Para asegurar la interoperabilidad de los servicios entre sub-sistemas diferentes se han creado varios mensajes estandarizados:

- Cooperative Awareness Message (CAM): mensaje periódico enviado de una a diez veces por segundo con datos de la posición, velocidad y dirección de los vehículos.
- Decentralised Enviromental Notification Message (DENM): mensaje de alerta enviado con alta prioridad y que informa de condiciones irregulares de la carretera, accidentes o aproximación de vehículos de emergencia.
- Signal Phase and Timing (SPAT): mensaje que proporciona información acerca del estado de los semáforos y sus cambios de estado futuros.
- MAP: mensaje que define la topología de un área (cruces, tipo de carriles, restricciones…)
- Service Announcement Message (SAM): mensaje que anuncia los servicios disponibles para el usuario, así como la tecnología de acceso de comunicación que se utiliza para acceder al servicio.
- DATEX II, TPEG, SAE J2375 (USA), Local Dynamic Map …

Capa Networking & Transport:

Esta capa incluye aplicaciones básicas de direccionamiento/encaminamiento y control de errores y flujo. Hay definidas 3 arquitecturas diferentes y todas permiten utilizan los protocolos IPv6 a nivel de red y TCP/UDP a nivel de transporte definidos por la IETF.

La arquitectura de la ETSI es GeoNetworking, la de la ISO es Communications Access for Land Mobiles (CALM) y el de IEEE es Wireless Access in Vehicular Environments (WAVE).

El GeoNetworking es un concepto desarrollado por organismos europeos y que busca una comunicación rápida y directa utilizando redes ad-hoc. Está formada por un protocolo de nivel de red, GeoNetworking; un protocolo de nivel de transporte, Basic Transport Protocol (BTP) y un mecanismo para transportar paquetes IPv6 sobre GeoNetworking.

El protocolo de nivel de red GeoNetworking tiene las siguientes características:

- Diseñado para comunicaciones móviles ad-hoc basadas en tecnologías inalámbricas de corto alcance.
- Permite comunicaciones en entornos móviles sin la necesidad de una infraestructura que coordine. Las comunicaciones son no orientadas a conexión y totalmente distribuidas. Pueden comunicarse vehículos i/o infraestructura de carretera.
- Permite la comunicación multi-salto, donde nodos de la red retransmiten los paquetes de otros nodos para extender el radio de cobertura de la red.
- Utiliza la posición geográfica como dirección para diseminar y transportar la información. Puede enviar paquetes a un nodo por su posición o a múltiples nodos en un área geográfica.
- Sólo los nodos del área geográfica destino procesan los paquetes, los nodos intermedios fuera del área geográfica destino no los procesan.
- Los nodos no utilizan tablas de encaminamiento, toman decisiones en función de la posición geográfica se encuentran.
- Está diseñado para dar servicio a aplicaciones heterogéneas. Permite transmisiones periódicas a alta cadencia de mensajes de seguridad, la rápida diseminación de paquetes en regiones geográficas para avisos de emergencia, y el transporte de paquetes unicast para aplicaciones de Internet.
- Se están diseñando mecanismos para evitar la congestión. En particular el Decentralized Congestión Control (DCC) que utiliza información del nivel de acceso y del nivel de red.
- Permite 4 esquemas de retransmisión más la posibilidad de extenderlos vía IPv6.

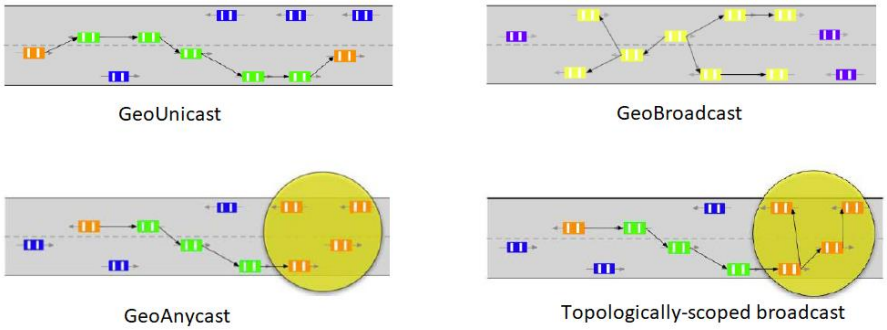

Figura 32: Tipos de retrnasmisión GeoNetworking [4]

- La posibilidad de interoperabilidad de IPv6 sobre el GeoNetworking permite utilizar la red IPv6 para comunicar zonas geográficas que no estén conectadas por vehículos.

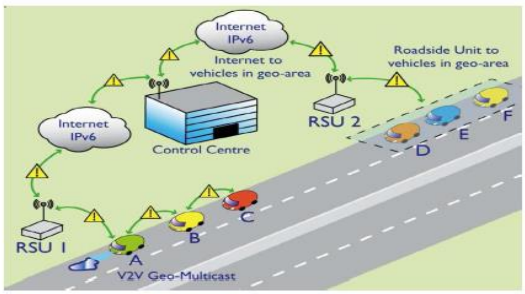

Figura 33: Retransmisión sin vehículos [4]

El protocolo de nivel de transporte Basic Transport Protocol (BTP) tiene las siguientes características:

- Proporciona un servicio de transporte extremo a extremo no orientado a conexión.
- Su propósito principal es multiplexar y demultiplexar mensajes de diferentes procesos de la capa de facilities ITS, para que puedan ser transmitidos sobre el protocolo de nivel de red GeoNetworking. La multiplexación está basada en puertos de 16 bits al estilo de TCP o UDP. También se ha copiado el modelo de puertos bien conocidos (well-known ports) para identificar aplicaciones específicas del nivel de facilities ITS.
- El BTP permite que las entidades de protocolo de nivel de facilites puedan acceder a los servicios del protocolo GeoNetworking y pasar información de control de protocolo entre la capa de facilities ITS y el protocolo GeoNetworking.
- El BTP es un protocolo muy ligero. Tiene una cabecera de 4 bytes y requiere una capacidad de proceso mínimo. Los 4 bytes pueden utilizarse para ubicar los dos puertos involucrados en la comunicación, o únicamente el puerto destino con dos bytes de información adicional.
- No proporciona un servicio fiable de transporte de paquetes. Es decir, los paquetes pueden perderse, llegar fuera de orden, duplicados o con errores. No hay campo de detección o corrección de errores en ninguno de los dos protocolos (ni GeoNetworking ni BTP). Se asume que las entidades que utilizan este protocolo son tolerables a estas situaciones o ellos mismos proporcionan la fiabilidad que requieren.

Capa Access Technologies:

Esta capa proporciona los niveles Medium Access Control (MAC) y físico de diferentes tipos de tecnologías. Los organismos estandarizadores quieren que las tecnologías C-ITS se puedan utilizar sobre cualquier tecnología radio. Se distinguen:

- 4 tecnologías radio principales: DSRC a 5 Ghz, redes celulares (2,5G, 3G, 4G/LTE), comunicaciones vía satélite y difusión radio (FM/DAB). Pero actualmente, las únicas que ofrecen retardos suficientemente bajos para aplicaciones de seguridad son DSRC y LTE.
- Otras tecnologías radio que debido a ciertas restricciones solo pueden ser válidas en ciertos escenarios: IEEE a/b/g/n, WiFi Direct, comunicaciones en frecuencias de infrarrojos, comunicaciones de baja velocidad y consumo (SIGFOX, LORA, IEEE, Bluetooth Low Energy).

La ISO solo ha estandarizado el uso de LTE y DSRC, la ETSI sistemas que operan en la banda de 5 Ghz i el IEE el 802.11p para WAVE.

ANEXO 2: Formatos archivos GPS

$GPRMC: recomended mínimum specific GPS/Transit data

$GPRMC,hhmmss.ss,A,llll.ll,a,yyyyy.yy,a,x.x,x.x,ddmmyy,x.x,a*hh												
fijo	1	2	3	4	5	6	7	8	9	10	11	12

Tabla 6: Campos GPRMC

1: tiempo en UTC de la posición

2: estado de la información {A = OK, V = aviso}

3: latitud en formato ll grados ll.ll minutos

4: {N, S} de la latitud

5: longitud en formato yy grados yy.yy minutos

6: {E, W} de la longitud

7: velocidad en nudos (SOG)

8: dirección en grados tomando como referencia 0º = norte

9: fecha en formato ddmmyy

10: declinación magnética en grados

11: E o W de la declinación

12: checksum

$GPGGA: global positioning system fix data

$GPGGA,hhmmss.ss,llll.ll,a ,yyyyy.yy, a, x, xx, x.x, x.x, M, x.x, M, x.x, xxxx*hh															
fijo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Tabla 7: Campos GPGGA

1: tiempo en UTC de la posición

2: latitud en formato ll grados ll.ll minutos

3: {N, S} de la latitud

4: longitud en formato yy grados yy.yy minutos

5: {E, W} de la longitud

6: indicador calidad del GPS: {0 = no válido, 1 = GPS, 2 = differential GPS}

7: número de satélites en uso

8: Horizontal Dilution of Precision (HDOP), posición relativa de la posición horizontal

9: altura de la antena sobre el nivel del mar

10: unidad de altitud de la antena

11: separación geoidal (diferencia entre el elipsoide de tierra WGS-84 y nivel medio del mar)

12: unidad de separación geoidal

13: tiempo en segundos desde la última actualización DGPS

14: ID estación DGPS

15: cheksum

Formato $GPGSV: GPS satelilites in view

$GPGSV,3,1,11, 03,03,111,00, 04,15,270,00, 06,01,010,00, 13,06,292,00*74																				
fijo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Tabla 8: Campos GPGSV

**los espacios son para mostrar más claramente la separación, en el caso real no hay espacios en blanco

1: número total de mensajes en este ciclo

2: número del mensaje

3: número total de satélites disponibles

4: identificador space vehicle del pseudorandom noise number

5: elevación en grados, 90 máximo

6: azimut, grados desde el norte, entre 0 y 359

7: SNR

8-11: igual que 4-7

12-15: igual que 4-7

16-19: igual que 4-7

20: checksum

ANEXO 3: Guía Vanetza

Para la instalación y funcionamiento del Vanetza de Github es suficiente con el siguiente script que instala las librerías necesarias y el propio proyecto.

```
#!/bin/bash
sudo apt-get update
sudo apt-get upgrade
sudo apt install python3 python3-pip
sudo pip3 install --upgrade pip
sudo pip install scikit-build
sudo python3 -m pip install cmake
sudo apt install g++ libcrypto++-dev libgeographic-dev libboost-date-
time-dev libboost-program-options-dev libboost-serialization-dev
libboost-system-dev git -y
sudo apt install libgps-dev gpsd-clients python-gps -y

git clone https://github.com/riebl/vanetza
cd vanetza/
mkdir build && cd build
cmake ..
make

cmake -D BUILD_SOCKTAP=ON ..
make
```

El proyecto contiene tres programas dentro de socktap: hello, cam i bench-in. Para poder ejecutarlos sin la comanda sudo, se recomienda ejecutar el siguiente comando dentro de la carpeta build:

```
$sudo setcap cap_net_raw+ep bin/<app>;
```

Una vez se tiene todo instalado y con permisos falta iniciar la aplicación. Pero para que funciona la aplicación se necesita tener un GPS conectado al equipo, o simularlo con gpsfake. Para simular con gpsfake primero hay que asegurar que el daemon gpsd está apagado. Para ello:

```
systemctl is-active gpsd; systemctl is-active gpsd.socket
```

Si no están “inactive”:

```
sudo systemctl stop gpsd; sudo systemctl stop gpsd.socket
```

Una vez están parados, arrancamos gpsfake.

```
gpsfake –c 0.15 –P <port> <file>.gps
```

Finalmente, solo queda arrancar el programa deseado. Desde la carpeta build de vanetza:

```
bin/<app> –i <interfaz> –gpsd-host localhost –gpsd-port <port>
```

la interfaz depende del uso que se quiere del programa en ese momento. Si es para pruebas suele ser “lo” y si se quiere probar con otros dispositivos “wlan0” o “eth0”. Todas las interfaces se pueden ver con el comando “ifconfig”.

El puerto para el gps conectado es 2947 y si ese fuera el caso, no hace falta poner la opción –gpsd-port. Pero si se está simulando con gpsfake hay que poner el mismo puerto en ambos sitios.

La aplicación resultante de este trabajo ha sido hecha con el mismo esquema de compilación. Por tanto, tan solo hay que situarse en la carpeta build del proyecto (si no existe se crea una vacía) e introducir la parte final del script del Vanetza original:

```
cmake –D BUILD_SOCTAP=ON ..
```

```
make
```

De esta manera ya se tendrá el receptor instalado y se inicia igual que Vanetza.

Consejos/aplicaciones útiles

Hay una aplicación que tienen todos los sistemas Linux y sirve para leer las entradas del GPS es “gpsmon”.

Simplemente escribiendo:

`gpsmon <host>:<port>` aparece la siguiente pantalla:

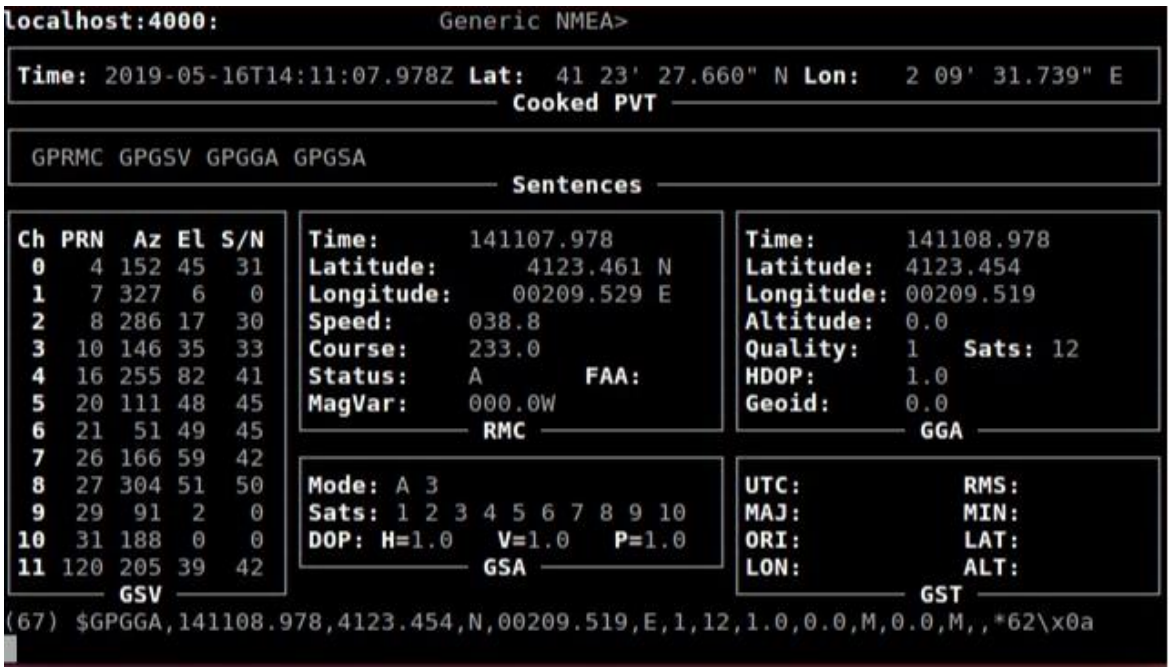

Figura 34: Ejemplo aplicación gpsmon

Para realizar distintas pruebas, interesa crear archivos GPS específicos. Para ello, a partir de archivos existentes se puede cambiar latitud, longitud, dirección y velocidad. Pero no es tan sencillo, porque al final de todas las líneas hay un checksun. Hay muchas aplicaciones para calcularlo, pero una rápida y sencilla se encuentra en la siguiente página web:

http://www.hhhh.org/wiml/proj/nmeaxor.html

Simplemente se copia la línea sin incluir el $ inicial ni el * final y te dice el valor correcto de checksun.

Si se quiere copiar un archivo GPS para utilizarlo en un futuro es sencillo. Se conecta el dispositivo al PC y se ve la salida que nos proporciona:

cat /dev/ttyACM0

A veces a problemas con la conexión y primero hay que introducir el siguiente comando si muestra líneas con formato "$TXT: ……."

sudo stty –F /dev/ttyACM0 -echo

Para copiar en un archivo solo queda:

Cat /dev/ttyACM0 >> <archivo>

Consejos para simularlo en Raspberry Pi

Si se quiere instalar el proyecto en una Raspberry a continuación se indica una manera de hacerlo desde cero y sin necesitar teclado y pantalla adicional para la Raspberry, solo con ssh desde otro ordenador y un bridge para conectarlos. Después una vez inicializado se puede conectar por una red inalámbrica también.

Primero hay que instalar en la tarjeta SD de las Raspberry la imagen de Raspbian. Una vez instalado, crear un archivo vacío llamado "ssh". Después, se introduce la tarjeta en la Raspberry y se conecta a la corriente eléctrica. También se conecta mediante cable ethernet un ordenador al bridge y la Raspberry con el bridge.

En la consola del ordenador introducir:

ping raspberrypi.local

De esta manera se logra conocer la dirección IP de la Raspberry.

Una vez se tiene la IP, introducir

ssh pi@<ip>

Pedirá contraseña para el usuario pi (introducido en la comanda ssh), es "raspberry".

Una vez se está dentro de la Raspberry ya se puede cambiar la contraseña o conectarla a una red inalámbrica para que sea todo más sencillo.

Para pasar un proyecto a una Raspberry se necesita comprimirlo y enviarlo por scp. A continuación, están los comandos necesarios:

Comprimir: tar –cvf <nombre_comprimido.tar> <carpeta_a_comprimir>

Pasar archivo de PC a RPI: scp <archivo_comprimido> pi@<ip>:<ruta_en_RPI_deseada>

Descomprimir: tar –xvf <archivo_comprimido>

ANEXO 4: Código herramienta web Javascript

Index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Turf Leaflet</title>

    <script src="turf.js" type="text/javascript" charset="utf-8"></script>

    <link rel="stylesheet" href="https://unpkg.com/leaflet@1.2.0/dist/leaflet.css" />

    <script src="https://unpkg.com/leaflet@1.2.0/dist/leaflet.js"></script>
    <script src='https://npmcdn.com/@turf/turf/turf.min.js'></script>

     <style>
    #map {
    width: 1000px;
     height: 600px; }
    </style>

</head>
    <body>

        First point latitude: <input type="number" id="point1-latitude">
        First point longitude: <input type="number" id="point1-longitude">
        First point bearing: <input type="number" id="point1-bearing"><br>
        Second point latitude: <input type="number" id="point2-latitude">
        Second point longitude: <input type="number" id="point2-longitude">
        Second point bearing: <input type="number" id="point2-bearing"><br>
        <input type="submit" value="Calcular punto intersección"
onclick="intersection()">

        <br><br>
        First point latitude: <input type="number" id="pointA-latitude">
        First point longitude: <input type="number" id="pointA-longitude"><br>
        Second point latitude: <input type="number" id="pointB-latitude">
        Second point longitude: <input type="number" id="pointB-longitude"><br>
        <input type="submit" value="Calcular distancia" onclick="distance()">

        <br><br>
        point latitude: <input type="number" id="point-latitude">
        point longitude: <input type="number" id="point-longitude"><br>
        bearing: <input type="number" id="bearing">
        distance: <input type="number" id="distance"><br>
        <input type="submit" value="Calcular punto" onclick="pointFollowingBearing()">
```

```
        <div id ="map"> </div>

        <div id ="resultado"></div>

    </body>
</html>
```

Turf.js

```
    function intersection(){
        var container = L.DomUtil.get('map');
            if(container != null){
                container._leaflet_id = null;
            }
        var map = L.map('map', {
            center: [41.39, 2.15],
        zoom: 12,
        });

    L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png').addTo(map);


    var point1 = turf.point([   Number(document.getElementById('point1-
longitude').value),
                                Number(document.getElementById('point1-
latitude').value)], {"marker-color": "F00"});
    var point2 = turf.point([   Number(document.getElementById('point2-
longitude').value),
                                Number(document.getElementById('point2-
latitude').value)], {"marker-color": "F00"});

    var distance = 1;//km
    var bearing = document.getElementById('point1-bearing').value;
    var options = {units: 'kilometers'};

    var point11 = turf.rhumbDestination(point1, distance, bearing,
options);

    bearing = document.getElementById('point2-bearing').value;

    var point22 = turf.rhumbDestination(point2, distance, bearing,
options);


    var linestring =
turf.lineString([  [point1.geometry.coordinates[0],point1.geometry.coordina
tes[1]],
                                        [point11.geometry.coordinates[0],po
int11.geometry.coordinates[1]]   ]);
    var linestring2 =
turf.lineString([ [point2.geometry.coordinates[0],point2.geometry.coordinat
es[1]],
                                        [point22.geometry.coordinates[0],po
int22.geometry.coordinates[1]]   ]);
```

```
    var intersection = turf.lineIntersect(linestring, linestring2);

        var geojson = [{
        "type": "Feature",
        "geometry": {
        "type": "Point",
        "coordinates": [point1.geometry.coordinates[0],
point1.geometry.coordinates[1] ]
                    }
        },{
        "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [point2.geometry.coordinates[0],
point2.geometry.coordinates[1] ]
                    }
        },{
        "type": "Feature",
        "geometry": {
        "type": "Point",
        "coordinates": [point11.geometry.coordinates[0],
point11.geometry.coordinates[1] ]
                    }
        },{
        "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [point22.geometry.coordinates[0],
point22.geometry.coordinates[1] ]
                    }
        },{
        "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates":
[intersection.features[0].geometry.coordinates[0],
intersection.features[0].geometry.coordinates[1] ]
                    }
        }
        ];

        coords = [];

            var puntos = L.geoJSON(geojson, {
                pointToLayer: function (feature, latlng) {
                        return L.marker(latlng);
                    },
```

```
                onEachFeature:  function (feature, layer) {
                coords.push(feature.geometry.coordinates);
                }
            });

        map.addLayer(puntos);


        L.geoJson(linestring, {color:"red"}).addTo(map);
        L.geoJson(linestring2, {color:"blue"}).addTo(map);

        document.getElementById('resultado').innerHTML = "Intersección: " +
intersection.features[0].geometry.coordinates[1].toFixed(8) + "--" +
intersection.features[0].geometry.coordinates[0].toFixed(8);
    }


    function distance(){
        var container = L.DomUtil.get('map');
            if(container != null){
                container._leaflet_id = null;
            }
            var map = L.map('map', {
                center: [41.39, 2.15],
            zoom: 12,
        });

    L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png').addTo(map);

    var pointA = turf.point([   Number(document.getElementById('pointA-
longitude').value),
                                Number(document.getElementById('pointA-
latitude').value)], {"marker-color": "F00"});
    var pointB = turf.point([   Number(document.getElementById('pointB-
longitude').value),
                                Number(document.getElementById('pointB-
latitude').value)], {"marker-color": "F00"});


    var linestring =
turf.lineString([  [pointA.geometry.coordinates[0],pointA.geometry.coordina
tes[1]],
                                        [pointB.geometry.coordinates[0],poi
ntB.geometry.coordinates[1]] ]);

    var options = {units: 'kilometers'};
    var distance = turf.distance(pointA, pointB, options)*1000;//meters
```

```
        var geojson = [{
        "type": "Feature",
        "geometry": {
        "type": "Point",
        "coordinates": [pointA.geometry.coordinates[0],
pointA.geometry.coordinates[1] ]
                    }
        },{
        "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [pointB.geometry.coordinates[0],
pointB.geometry.coordinates[1] ]
                    }
        }
        ];

        coords = [];

            var puntos = L.geoJSON(geojson, {
                pointToLayer: function (feature, latlng) {
                        return L.marker(latlng);
                    },
                onEachFeature:  function (feature, layer) {
                coords.push(feature.geometry.coordinates);
                }
            });

        map.addLayer(puntos);

        L.geoJson(linestring, {color:"red"}).addTo(map);

        document.getElementById("resultado").innerHTML = "distancia: " +
distance.toFixed(3) + "metros";
    }

    function pointFollowingBearing(){
        var container = L.DomUtil.get('map');
            if(container != null){
                container._leaflet_id = null;
            }
        var map = L.map('map', {
            center: [41.39, 2.15],
        zoom: 12,
        });

    L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png').addTo(map);
```

```
    var point = turf.point([   Number(document.getElementById('point-
longitude').value),
                            Number(document.getElementById('point-
latitude').value)], {"marker-color": "F00"});

    var distance = Number(document.getElementById('distance').value);

    var bearing = document.getElementById('bearing').value;
    var options = {units: 'kilometers'};

    var point2 = turf.rhumbDestination(point, distance, bearing, options);


    var linestring =
turf.lineString([  [point.geometry.coordinates[0],point.geometry.coordinate
s[1]],
                                        [point2.geometry.coordinates[0],poi
nt2.geometry.coordinates[1]] ]);


        var geojson = [{
        "type": "Feature",
        "geometry": {
        "type": "Point",
        "coordinates": [point.geometry.coordinates[0],
point.geometry.coordinates[1] ]
                    }
        },{
        "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [point2.geometry.coordinates[0],
point2.geometry.coordinates[1] ]
                    }
        }
        ];

        coords = [];

            var puntos = L.geoJSON(geojson, {
                pointToLayer: function (feature, latlng) {
                        return L.marker(latlng);
                    },
                onEachFeature:  function (feature, layer) {
                coords.push(feature.geometry.coordinates);
                }
            });
```

```
        map.addLayer(puntos);

        L.geoJson(linestring, {color:"red"}).addTo(map);

        document.getElementById('resultado').innerHTML = "punto: " +
point2.geometry.coordinates[1].toFixed(8) + "--" +
point2.geometry.coordinates[0].toFixed(8);
    }
```

ANEXO 5: Código del programa

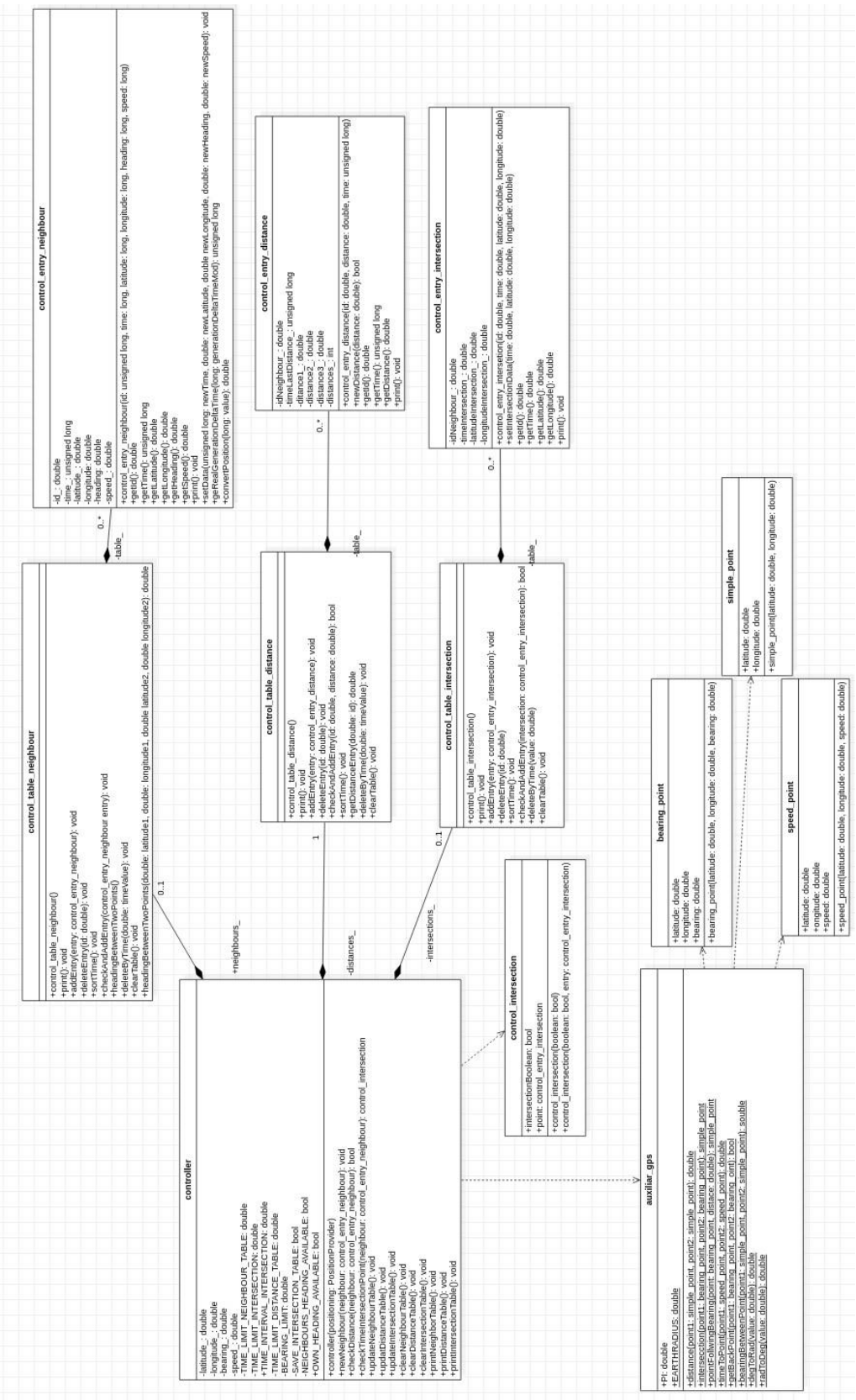

Figura 35: Diagrama UML del programa

control_entry_neighbour.h

```
class control_entry_neighbour{

    private:
        double id_;
        unsigned long time_;
        double latitude_;
        double longitude_;
        double heading_;
        double speed_;

    public:
        control_entry_neighbour(unsigned long idNeighbour, long timeNeighbour, long
latitudeNeighbour,
                        long longitudeNeighbour, long headingNeighbour, long speedNeighbour);

        /**
         * @brief Getter
         * @return id_
         */
        double getId();
        /**
         * @brief Getter
         * @return time_
         */
        unsigned long getTime();
        /**
         * @brief Getter
         * @return latitude_
         */
        double getLatitude();
        /**
         * @brief Getter
         * @return longitude_
         */
        double getLongitude();
        /**
         * @brief Getter
         * @return heading_
         */
        double getHeading();
        /**
         * @brief Getter
         * @return speed_
         */
        double getSpeed();
         /**
```

```
         * @brief Setter
         * @param newTime
         * @param newLatitude
         * @param newLongitude
         * @param newHeading
         * @param newSpeed
         */
        void setData( unsigned long newTime, double newLatitude, double newLongitude, double
newHeading, double newSpeed );
        /**
         * @brief Write by console the neighbour attributes
         */
        void print();
        /**
         * @brief Gets the complete time from the wrapped time
         * @param generationDeltaTimeMod
         * @return completeTime
         */
        unsigned long getRealGenerationDeltaTime(long generationDeltaTimeMod);
         /**
         * @brief Modifies the value to convert it to the desired units
         * @param value
         * @return convertedValue
         */
        double convertPosition(long value);

};
```

control_entry_neighbour.cpp

```
#include <iostream>
#include <chrono>
#include <iomanip>

#ifndef CONTROL_ENTRY_NEIGHBOUR_H
#define CONTROL_ENTRY_NEIGHBOUR_H
#include "control_entry_neighbour.h"
#endif

control_entry_neighbour::control_entry_neighbour(   unsigned long idNeighbour, long
timeNeighbour, long latitudeNeighbour,
                        long longitudeNeighbour, long headingNeighbour, long
speedNeighbour):
                        id_(idNeighbour),
time_(getRealGenerationDeltaTime(timeNeighbour)), heading_(headingNeighbour),
speed_(speedNeighbour){
                            speed_ = speedNeighbour / 100;
                            latitude_ = convertPosition (latitudeNeighbour);
                            longitude_ = convertPosition (longitudeNeighbour);

                        }

double control_entry_neighbour::getId(){ return id_;}

unsigned long control_entry_neighbour::getTime(){ return time_;}

double control_entry_neighbour::getLatitude(){ return latitude_;}

double control_entry_neighbour::getLongitude(){ return longitude_;}

double control_entry_neighbour::getHeading(){ return heading_;}

double control_entry_neighbour::getSpeed(){ return speed_;}

void control_entry_neighbour::setData( unsigned long newTime, double newLatitude, double
newLongitude, double newHeading, double newSpeed ){

    time_ = newTime;
    latitude_ = newLatitude;
    longitude_ = newLongitude;
    heading_ = newHeading;
    speed_ = newSpeed;

}

void control_entry_neighbour::print(){
```

```
    std::cout << std::setprecision(10)      << "id: "       << id_
                    << " || time: "     << time_
                    << " || latitude: " << latitude_
                    << " || longitude: " << longitude_
                    << " || heading: "  << heading_
                    << " || speed: "    << speed_      << std::endl;
}

unsigned long control_entry_neighbour::getRealGenerationDeltaTime(long
generationDeltaTime_Mod){

    using namespace std::chrono;
    milliseconds currentTime_ms = duration_cast< milliseconds
>(system_clock::now().time_since_epoch());
    uint16_t currentTime_ms_mod = currentTime_ms.count();

    long difference_time;

    if(generationDeltaTime_Mod <= currentTime_ms_mod){

        difference_time = currentTime_ms_mod - generationDeltaTime_Mod;

    }else{

        difference_time = currentTime_ms_mod + (65535 - generationDeltaTime_Mod);

    }

    return currentTime_ms.count() - difference_time;

}

double control_entry_neighbour::convertPosition(long value){


    double out = value * 0.0000001;

    return out;

}
```

control_entry_distance.h

```
class control_entry_distance{

    private:
        double idNeighbour_;
        unsigned long timeLastDistance_;
        double distance1_;
        double distance2_;
        double distance3_;
        int distances_;
    public:
        control_entry_distance(double id, double distance, unsigned long time);

        /**
         * @brief Updates the neighbour's distance in the table
         * @param distance
         * @param time
         * @return Boolean that indicates if the neighbour is approaching
         */
        bool newDistance(double distance, unsigned long time);
        /**
         * @brief Getter
         * @return idNeighbour
         */
        double getId();
        /**
         * @brief Getter
         * @return timeLastDistance
         */
        unsigned long getTime();
        /**
         * @brief Getter
         * @return last distance
         */
        double getDistance();
        /**
         * @brief Write by console the distance attributes
         */
        void print();

};
```

control_entry_distance.cpp

```
#include <iostream>

#ifndef CONTROL_ENTRY_DISTANCE_H
#define CONTROL_ENTRY_DISTANCE_H
#include "../entries/control_entry_distance.h"
#endif

using namespace std;


control_entry_distance::control_entry_distance(double idEntry, double distanceEntry,
unsigned long timeEntry):
                                                idNeighbour_(idEntry),
distance1_(distanceEntry), distances_(1), timeLastDistance_(timeEntry)
                                                {};

bool control_entry_distance::newDistance(double distanceEntry, unsigned long timeEntry){

    timeLastDistance_ = timeEntry;

    switch(distances_){
        case 1:
            distance2_ = distanceEntry;
            distances_++;
            return false;
        case 2:
            distance3_ = distanceEntry;
            distances_++;
            if(distance1_ >= distance2_ && distance2_ >= distance3_){
                return true;
            }else{
                return false;
            }
        case 3:
            distance1_ = distance2_;
            distance2_ = distance3_;
            distance3_ = distanceEntry;
            if(distance1_ >= distance2_ && distance2_ >= distance3_){
                return true;
            }else{
                return false;
            }
    }
}

double control_entry_distance::getId(){
```

```
    return idNeighbour_;
}

unsigned long control_entry_distance::getTime(){
    return timeLastDistance_;
}

double control_entry_distance::getDistance(){
    switch(distances_){
        case 1:
            return distance1_;
        case 2:
            return distance2_;
        case 3:
           return distance3_;
    }
}

void control_entry_distance::print(){

    cout    << "id: " << idNeighbour_
            << " || time: " << timeLastDistance_
            << " || distances: " << distances_
            << " || distance1: " << distance1_
            << " || distance2: " << distance2_
            << " || distance3: " << distance3_ << endl;
}
```

control_entry_neighbour.h

```
class control_entry_intersection{

    private:
        double idNeighbour_;
        double timeIntersection_;
        double latitudeIntersection_;
        double longitudeIntersection_;

    public:
        control_entry_intersection(double id, double time, double latitude, double
longitude);
        /**
         * @brief Updates the intersection table
         * @param time
         * @param latitude
         * @param longitude
         */
        void setIntersectionData(double time, double latitude, double longitude);
        /**
         * @brief Getter
         * @return idNeighbour
         */
        double getId();
        /**
         * @brief Getter
         * @return timeIntersection
         */
        double getTime();
        /**
         * @brief Getter
         * @return latitudeIntersection
         */
        double getLatitude();
        /**
         * @brief Getter
         * @return longitudeIntersection
         */
        double getLongitude();
        /**
         * @brief Write by console the intersection attributes
         */
        void print();

};
```

control_entry_neighbour.cpp

```
#include <iostream>
#include <chrono>
#include <math.h>

#ifndef CONTROL_ENTRY_INTERSECTION_H
#define CONTROL_ENTRY_INTERSECTION_H
#include "control_entry_intersection.h"
#endif

using namespace std;
using namespace std::chrono;

control_entry_intersection::control_entry_intersection(double id, double time, double
latitude, double longitude):

idNeighbour_(id),timeIntersection_(time),latitudeIntersection_(latitude),
longitudeIntersection_(longitude) {};

void control_entry_intersection::setIntersectionData(double time, double latitude,
double longitude){

    timeIntersection_ = time;
    latitudeIntersection_ = latitude;
    longitudeIntersection_ = longitude;

}

double control_entry_intersection::getId(){return idNeighbour_;}

double control_entry_intersection::getTime(){return timeIntersection_;}

double control_entry_intersection::getLatitude(){return latitudeIntersection_;}

double control_entry_intersection::getLongitude(){return longitudeIntersection_;}

void control_entry_intersection::print(){


    milliseconds timeNow = duration_cast< milliseconds
>(system_clock::now().time_since_epoch());

    cout << "id: " << idNeighbour_
         << " || time intersection: " << (timeIntersection_ - timeNow.count()) / 1000
         << " || latitude intersection: " << latitudeIntersection_
         << " || longitude intersection: " << longitudeIntersection_ << endl;
}
```

control_table_neighbour.h

```
#include <list>

#ifndef CONTROL_ENTRY_NEIGHBOUR_H
#define CONTROL_ENTRY_NEIGHBOUR_H
#include "../entries/control_entry_neighbour.h"
#endif

class control_table_neighbour{

    private:
        std::list<control_entry_neighbour> table_;

    public:
        control_table_neighbour();
        /**
         * @brief print by console each table entry
         */
        void print();
        /**
         * @brief Adds to the table the entry
         * @param control_entry_neighbour
         */
        void addEntry(control_entry_neighbour);
        /**
         * @brief Deletes from the table the entry with the indicated id
         * @param id
         */
        void deleteEntry(double id);
        /**
         * @brief Checks if the entry exists in the table: if exists, it modifies; if it
does not exist, it adds to the table
         * @param control_entry_neighbour
         */
        void checkAndAddEntry(control_entry_neighbour& entry);
        /**
         * @brief orders the table according to the time of its elements
         */
        void sortTime();
        /**
         * @brief Deletes all values from the table with a longer time than indicated
         * @param timeValue
         */
        void deleteByTime(double timeValue);
        /**
```

```
         * Deletes all entries in the table
         */
        void clearTable();

        /**
         * @brief Calculates the heading between two points from its respective latitude
and longitude
         * @param latitude1
         * @param longitude1
         * @param latitude2
         * @param longitude2
         * @return heading
         */
        double headingBetweenTwoPoints( double latitude1, double longitude1, double
latitude2, double longitude2);
};
```

control_table_neighbour.cpp

```
#ifndef CONTROL_TABLE_NEIGHBOUR_H
#define CONTROL_TABLE_NEIGHBOUR_H
#include "control_table_neighbour.h"
#endif

#include <iostream>
#include <algorithm>
#include <chrono>
#include <iomanip>


using namespace std;

control_table_neighbour::control_table_neighbour(){}

void control_table_neighbour::print(){
std::cout << " neighbours table: " << std::endl;
    for (std::list<control_entry_neighbour>::iterator it = table_.begin(); it !=
table_.end(); ++it){
            it->print();
    }
}

void control_table_neighbour::addEntry(control_entry_neighbour ce){
    table_.push_front(ce);
}

void control_table_neighbour::deleteEntry(double id){
    std::list<control_entry_neighbour>::iterator it_deleteEntry;
    for (std::list<control_entry_neighbour>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        if ( it->getId() == id){
          it_deleteEntry = it;
        }
    }
    table_.erase(it_deleteEntry);
}

void control_table_neighbour::checkAndAddEntry(control_entry_neighbour& entry){
    bool add = true;
    for (std::list<control_entry_neighbour>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        if ( it->getId() == entry.getId()){
            double heading = headingBetweenTwoPoints( it->getLatitude(), it-
>getLongitude(), entry.getLatitude(), entry.getLongitude());
```

```
            it->setData(entry.getTime(), entry.getLatitude(), entry.getLongitude(),
heading, entry.getSpeed());
            entry.setData(entry.getTime(), entry.getLatitude(), entry.getLongitude(),
heading, entry.getSpeed());
            add = false;
            break;
        }
    }
    if(add){
        addEntry(entry);
    }
}

bool compare_time (control_entry_neighbour& first, control_entry_neighbour& second){
  return first.getTime() < second.getTime();
}

void control_table_neighbour::sortTime(){
    table_.sort(compare_time);
}

void control_table_neighbour::deleteByTime(double timeValue){
    sortTime();
    std::list<control_entry_neighbour>::iterator itFirst, itLast;
    bool deleteEntryControl=false;
    using namespace std::chrono;
    milliseconds currentTime = duration_cast< milliseconds
>(system_clock::now().time_since_epoch());
    for (std::list<control_entry_neighbour>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        unsigned long differenceTime = currentTime.count() - it->getTime();
        if(!deleteEntryControl && ( differenceTime < timeValue ) ){
            break;
        }
        if(deleteEntryControl &&  ( differenceTime > timeValue ) ){
            itLast = it;
        }
        if(!deleteEntryControl &&( differenceTime > timeValue ) ){
            deleteEntryControl = true;
            itFirst = itLast = it;
        }
    }
    if(deleteEntryControl){
        table_.erase(itFirst, ++itLast);
    }
}

void control_table_neighbour::clearTable(){
```

```
    table_.clear();
}

double control_table_neighbour::headingBetweenTwoPoints( double latitude1, double
longitude1, double latitude2, double longitude2){

        double lat1 = (3.14/180) * latitude1;
        double lon1 = (3.14/180) * longitude1;
        double lat2 = (3.14/180) * latitude2;
        double lon2 = (3.14/180) * longitude2;

        double y = sin(lon2 - lon1 ) * cos( lat2 );
        double x = cos( lat1 ) * sin( lat2 ) - sin( lat1 ) * cos( lat2 ) * cos ( lon2 -
lon1 );

        double bearingRad = atan2( y , x );
        double bearingDeg = (180/3.14) * (bearingRad);

        return std::fmod(bearingDeg + 360, 360);
}
```

control_table_distance.h

```
#include <list>

#ifndef CONTROL_ENTRY_DISTANCE_H
#define CONTROL_ENTRY_DISTANCE_H
#include "../entries/control_entry_distance.h"
#endif

class control_table_distance{

    private:
        std::list<control_entry_distance> table_;

    public:
        control_table_distance();
         /**
         * @brief Write by console the table
         */
        void print();
        /**
         * @brief Adds to the table the entry
         * @param control_entry_distance
         */
        void addEntry(control_entry_distance);
        /**
         * @brief Deletes from the table the entry with the indicated id
         * @param id
         */
        void deleteEntry(double id);
        /**
         * @brief Checks if the entry exists in the table: if exists, it modifies; if it
does not exist, it adds to the table
         * @param control_entry_neighbour
         */
        bool checkAndAddEntry(double id, double distance, double time);
        /**
         * @brief orders the table according to the time of its elements
         */
        void sortTime();
        /**
         * @brief Deletes all values from the table with a longer time than indicated
         * @param timeValue
         */
        void deleteByTime(double timeValue);
        /**
         * Deletes all entries in the table
         */
```

```
        void clearTable();

        /**
         * @brief Returns the last distance of the entry with indicated id
         * @param id
         * @return distance
         */
        double getDistanceEntry(double id);

};
```

control_table_distance.cpp

```
#ifndef CONTROL_TABLE_DISTANCE_H
#define CONTROL_TABLE_DISTANCE_H
#include "control_table_distance.h"
#endif

#include <iostream>
#include <algorithm>
#include <chrono>
#include <iomanip>

using namespace std;

control_table_distance::control_table_distance(){}

void control_table_distance::print(){
    std::cout << " distances table:" << std::endl;
    for (std::list<control_entry_distance>::iterator it = table_.begin(); it !=
table_.end(); ++it){
            it->print();
        }
}

void control_table_distance::addEntry(control_entry_distance ce){
    table_.push_front(ce);
}

void control_table_distance::deleteEntry(double id){
    std::list<control_entry_distance>::iterator it_deleteEntry;
    for (std::list<control_entry_distance>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        if ( it->getId() == id){
            it_deleteEntry = it;
        }
    }
    table_.erase(it_deleteEntry);
}

bool control_table_distance::checkAndAddEntry(double idEntry, double distanceEntry,
double timeEntry){
    for (std::list<control_entry_distance>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        if ( it->getId() == idEntry){
            return (it->newDistance(distanceEntry,timeEntry));
        }
    }
```

```
    control_entry_distance new_control_entry_distance(idEntry, distanceEntry,
timeEntry);
    addEntry(new_control_entry_distance);
    return false;
}

bool compare_time (control_entry_distance& first, control_entry_distance& second){
    return first.getTime() > second.getTime();
}

void control_table_distance::sortTime(){
    table_.sort(compare_time);
}

void control_table_distance::deleteByTime(double timeValue){
    sortTime();
    std::list<control_entry_distance>::iterator itFirst, itLast;
    bool deleteEntryControl=false;
    using namespace std::chrono;
    milliseconds currentTime = duration_cast< milliseconds
>(system_clock::now().time_since_epoch());
    for (std::list<control_entry_distance>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        unsigned long differenceTime = currentTime.count() - it->getTime();
        if(!deleteEntryControl && ( differenceTime < timeValue ) ){
            break;
        }
        if(deleteEntryControl &&  ( differenceTime > timeValue ) ){
            itLast = it;
        }
        if(!deleteEntryControl &&( differenceTime > timeValue ) ){
            deleteEntryControl = true;
            itFirst = itLast = it;
        }
    }
    if(deleteEntryControl){
        table_.erase(itFirst, ++itLast);
    }
}

void control_table_distance::clearTable(){
    table_.clear();
}

double control_table_distance::getDistanceEntry(double id){
    for (std::list<control_entry_distance>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        if ( it->getId() == id){
```

```
            return (it->getDistance());
        }
    }
}
```

control_table_intersection.h

```
#include <list>

#ifndef CONTROL_ENTRY_INTERSECTION_H
#define CONTROL_ENTRY_INTERSECTION_H
#include "../entries/control_entry_intersection.h"
#endif

class control_table_intersection{

    private:
        std::list<control_entry_intersection> table_;

    public:
        control_table_intersection();
        /**
         * @brief print by console the table
         */
        void print();
        /**
         * @brief Adds to the table the entry
         * @param control_entry_intersection
         */
        void addEntry(control_entry_intersection);
        /**
         * @brief Deletes from the table the entry with the indicated id
         * @param id
         */
        void deleteEntry(double id);
        /**
         * @brief Checks if the entry exists in the table: if exists, it modifies; if it does not
exist, it adds to the table
         * @param control_entry_neighbour
         */
        void checkAndAddEntry(control_entry_intersection);
        /**
         * @brief orders the table according to the time of its elements
         */
        void sortTime();
        /**
         * @brief Deletes all values from the table with a longer time than indicated
         * @param timeValue
         */
        void deleteByTime();
        /**
         * Deletes all entries in the table
         */
```

```
        void clearTable();

};
```

control_table_intersection.cpp

```
#ifndef CONTROL_TABLE_INTERSECTION_H
#define CONTROL_TABLE_INTERSECTION_H
#include "control_table_intersection.h"
#endif

#include <iostream>
#include <algorithm>
#include <chrono>



using namespace std;

control_table_intersection::control_table_intersection(){}

void control_table_intersection::print(){
    std::cout << " intersections table: " << std::endl;
    for (std::list<control_entry_intersection>::iterator it = table_.begin(); it !=
table_.end(); ++it){
            it->print();
    }
}

void control_table_intersection::addEntry(control_entry_intersection ce){
    table_.push_front(ce);
}

void control_table_intersection::deleteEntry(double id){
    std::list<control_entry_intersection>::iterator it_deleteEntry;
    for (std::list<control_entry_intersection>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        if ( it->getId() == id){
          it_deleteEntry = it;
        }
    }
    table_.erase(it_deleteEntry);
}

void control_table_intersection::checkAndAddEntry(control_entry_intersection entry){
    bool add = true;
    for (std::list<control_entry_intersection>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        if ( it->getId() == entry.getId()){
            it->setIntersectionData(entry.getTime(), entry.getLatitude(),
entry.getLongitude());
```

```
            add = false;
            break;
        }
    }
    if(add){
        addEntry(entry);
    }
}

bool compare_time (control_entry_intersection& first, control_entry_intersection&
second){
  return first.getTime() > second.getTime();
}

void control_table_intersection::sortTime(){
    table_.sort(compare_time);
}

void control_table_intersection::deleteByTime(){
    sortTime();
    std::list<control_entry_intersection>::iterator itFirst, itLast;
    bool deleteEntryControl=false;
    using namespace std::chrono;
    milliseconds timeNow = duration_cast< milliseconds
>(system_clock::now().time_since_epoch());
    for (std::list<control_entry_intersection>::iterator it = table_.begin(); it !=
table_.end(); ++it){
        if(!deleteEntryControl){
            if(it->getTime() > timeNow.count() ){
                break;
            }else{
                deleteEntryControl = true;
                itFirst = itLast = it;
            }
        }else{
            if(it->getTime() < timeNow.count() ){
                itLast = it;
            }else{
                break;
            }
        }
    }
    if(deleteEntryControl){
        table_.erase(itFirst, ++itLast);
    }
}

void control_table_intersection::clearTable(){
```

```
    table_.clear();
}
```

auxiliar_gps.h

```
#include <cmath>
#include <gps.h>

class auxiliar_gps{

    public:
    /**
     * @brief point with latitude and longitude
     * @param latitude
     * @param longitude
     */
    struct simple_point{
        double latitude;
        double longitude;
        simple_point(double lat,double lon):latitude(lat), longitude(lon){}
        simple_point():latitude(), longitude(){}
    };

    /**
     * @brief point with latitude, longitude and bearing
     * @param latitude
     * @param longitude
     * @param bearing
     */
    struct bearing_point{
        double latitude;
        double longitude;
        double bearing;
        bearing_point(double lat, double lon, double bea): latitude(lat),
longitude(lon), bearing(bea){}
    };

    /**
     * @brief point with latitude, longitude and bearing
     * @param latitude
     * @param longitude
     * @param speed
     */
    struct speed_point{
        double latitude;
        double longitude;
        double speed;
        speed_point(double lat, double lon, double sp): latitude(lat), longitude(lon),
speed(sp){}
    };
```

```
    auxiliar_gps(void);

    /**
     * @brief Calculates the distance betweeen to initial positions
     * @param point1 One simple point that includes latitude and longitude
     * @param point2 The other simple point that includes latitude and longitude
     * @return The distance between the two points in meters
     */
    static double distance(simple_point point1, simple_point point2);

    /**
     * @brief Calculates the point where two trajectories intersect
     * @param point1 One bearing point
     * @param point2 The other bearing point
     * @return The simple point where the trajectories intersect
     */
    static simple_point intersection(bearing_point point1, bearing_point point2);

    /**
     * @brief Calculates a destination simple point based on a starting position, a
heading and a distance
     * @param point1 Bearing point that includes initial position and heading
     * @param point2 Distance in meters
     * @return The destination simple point
     */
    static simple_point pointFollowingBearing(bearing_point point, double distance);

    /**
     * @brief Calculates the time to reach a point based on the current simple point and
current speed
     * @param point1 Speed point that includes the initial position and the speed
     * @param point2 Destination simple point
     * @return The time to reach the destination point in seconds
     */
    static double timeToPoint(speed_point point1, simple_point point2);

    /**
     * @brief Calculates what point is back from other
     * @param point1 one bearing point
     * @param point2 The other bearing point
     * @return True if point 2 is back; False if point1 is back
     */
    static bool getBackPoint(bearing_point point1, bearing_point point2);

    /**
     * @brief Calculates the bearing between the two points
     * @param point1 one simple point
     * @param point2 The other simple point
```

```
     * @return bearingValue
     */
    static double bearingBetweenPoints( simple_point point1, simple_point point2);
    /**
     * @brief Transfroms degress into radians
     * @param degrees
     * @return radians
     */
    static double degToRad(double degrees);

    /**
     * @brief Transfroms radians into degrees
     * @param radians
     * @return degrees
     */
    static double radToDeg(double radians);

    static constexpr double PI = GPS_PI;
    static constexpr double EARTHRADIUS = WGS84A;//METERS




};
```

auxiliar_gps.cpp

```
#include "auxiliar_gps.h"
#include <cmath>
   /* abs */
#include <algorithm> // for std::min
#include <iostream>  // std::cout
#include <iomanip>


using namespace std;

    auxiliar_gps::auxiliar_gps(void)
    {
    }

    double auxiliar_gps::distance(auxiliar_gps::simple_point point1,
auxiliar_gps::simple_point point2){

        double lat1 = degToRad(point1.latitude);
        double lon1 = degToRad(point1.longitude);
        double lat2 = degToRad(point2.latitude);
        double lon2 = degToRad(point2.longitude);

        double a= acos(sin(lat1)*sin(lat2) + double(cos(lat1)*cos(lat2)*cos(lon2-
lon1)));

        return a*auxiliar_gps::EARTHRADIUS;

    }

    auxiliar_gps::simple_point auxiliar_gps::intersection(auxiliar_gps::bearing_point
point1 , auxiliar_gps::bearing_point point2){

        double  latitudeDiff, longitudeDiff, angularDist, bearingStart, bearingFinal,
bearing12, bearing21,
                ang1, ang2, ang3, angularDist13, lat3, incrementlongitude13, lon3;

        double latitude1 = degToRad(point1.latitude);
        double latitude2 = degToRad(point2.latitude);
        double longitude1 = degToRad(point1.longitude);
        double longitude2 = degToRad(point2.longitude);
        double bearing1 = degToRad(point1.bearing);
        double bearing2 = degToRad(point2.bearing);

        latitudeDiff = latitude2-latitude1;
        longitudeDiff = longitude2-longitude1;
```

```
        angularDist = 2*asin(sqrt(pow(sin(latitudeDiff/2),2) + cos(latitude1) *
cos(latitude2) * pow(sin(longitudeDiff/2),2)));

        bearingStart = acos((sin(latitude2)-sin(latitude1)*cos(angularDist)) /
(sin(angularDist)*cos(latitude1)));
        bearingFinal = acos((sin(latitude1)-sin(latitude2)*cos(angularDist)) /
(sin(angularDist)*cos(latitude2)));

        if(sin(longitude2-longitude1)>0){
            bearing12 = bearingStart;
            bearing21 = 2*auxiliar_gps::PI - bearingFinal;
        }else{
            bearing12 = 2*auxiliar_gps::PI - bearingStart;
            bearing21 = bearingFinal;
        }

        ang1 = bearing1 - bearing12;
        ang2 = bearing21 - bearing2;
        ang3 = acos(-cos(ang1)*cos(ang2) + sin(ang1)*sin(ang2)*cos(angularDist));

        angularDist13 =
atan2(sin(angularDist*sin(ang1)*sin(ang2)),cos(ang2)+cos(ang1)*cos(ang3));

        lat3 =
asin(sin(latitude1)*cos(angularDist13)+cos(latitude1)*sin(angularDist13)*cos(bearing1));

        incrementlongitude13 =
atan2(sin(bearing1)*sin(angularDist13)*cos(latitude1),cos(angularDist13)-
sin(latitude1)*sin(lat3));

        lon3 = longitude1 + incrementlongitude13;

        auxiliar_gps::simple_point intersection =
auxiliar_gps::simple_point(radToDeg(lat3),radToDeg(lon3));

        return intersection;
    }

    auxiliar_gps::simple_point auxiliar_gps::pointFollowingBearing(
auxiliar_gps::bearing_point point, double distance){

        double latitudeStart = degToRad(point.latitude);
        double longitudeStart = degToRad(point.longitude);
        double bearing = degToRad(point.bearing);

        double latitude = latitudeStart + (degToRad(distance/110540)*cos(bearing));
```

```
        double longitude = longitudeStart +
(degToRad(distance/(111320*cos(latitudeStart)))*sin(bearing));

        auxiliar_gps::simple_point intersection =
auxiliar_gps::simple_point(radToDeg(latitude), radToDeg(longitude));

        return intersection;
    }

    double auxiliar_gps::timeToPoint(auxiliar_gps::speed_point point1,
auxiliar_gps::simple_point point2){

        auxiliar_gps::simple_point point = auxiliar_gps::simple_point(point1.latitude,
point1.longitude);
        double distance_to_point = auxiliar_gps::distance(point, point2);

        return distance_to_point / point1.speed;
    }

    bool auxiliar_gps::getBackPoint(auxiliar_gps::bearing_point point1,
auxiliar_gps::bearing_point point2){

        if(point1.bearing >= 45 && point1.bearing < 135){

            if(point1.longitude < point2.longitude){
                return true;
            }else{
                return false;
            }

        }else if(point1.bearing >= 135 && point1.bearing < 225){

            if(point1.latitude > point2.latitude){
                return true;
            }else{
                return false;
            }

        }else if(point1.bearing >= 225 && point1.bearing < 315){

            if(point1.longitude > point2.longitude){
                return true;
            }else{
                return false;
            }

        }else{
```

```
            if(point1.latitude < point2.latitude){
                return true;
            }else{
                return false;
            }

        }

    }


    double auxiliar_gps::bearingBetweenPoints( simple_point point1, simple_point
point2){

        double lat1 = degToRad(point1.latitude);
        double lon1 = degToRad(point1.longitude);
        double lat2 = degToRad(point2.latitude);
        double lon2 = degToRad(point2.longitude);

        double y = sin(lon2 - lon1 ) * cos( lat2 );
        double x = cos( lat1 ) * sin( lat2 ) - sin( lat1 ) * cos( lat2 ) * cos ( lon2 -
lon1 );

        double bearingRad = atan2( y , x );
        double bearingDeg = radToDeg(bearingRad);

        return std::fmod(bearingDeg + 360, 360);

    }

    double auxiliar_gps::degToRad(double degrees){
        return degrees * DEG_2_RAD;
    }

    double auxiliar_gps::radToDeg(double radians){
        return radians * RAD_2_DEG;
    }
```

controller.h

```
#ifndef CONTROL_TABLE_NEIGHBOUR_H
#define CONTROL_TABLE_NEIGHBOUR_H
#include "tables/control_table_neighbour.h"
#endif

#ifndef CONTROL_TABLE_DISTANCE_H
#define CONTROL_TABLE_DISTANCE_H
#include "tables/control_table_distance.h"
#endif

#ifndef CONTROL_TABLE_INTERSECTION_H
#define CONTROL_TABLE_INTERSECTION_H
#include "tables/control_table_intersection.h"
#endif

#ifndef AUXILIAR_GPS_H
#define AUXILIAR_GPS_H
#include "auxiliar_gps.h"
#endif

#include <vanetza/common/position_provider.hpp>

class controller{

    private:
        control_table_neighbour neighbours_;
        control_table_distance distances_;
        control_table_intersection intersections_;
        double latitude_, longitude_, bearing_, speed_;
        double TIME_LIMIT_NEIGHBOUR_TABLE, TIME_LIMIT_DISTANCE_TABLE,
TIME_LIMIT_INTERSECTION_TABLE, TIME_INTERVAL_INTERSECTION, BEARING_LIMIT;
        bool SAVE_INTERSECTION_TABLE, NEIGHBOURS_HEADING_AVAILABLE,
OWN_HEADING_AVAILABLE;

        vanetza::PositionProvider& positioning_;
        void updateValues();
        double calculateBearing(double newLatitude, double newLongitude );

    public:
        controller(vanetza::PositionProvider& positioning);
        struct control_intersection{
            bool intersectionBoolean;
            control_entry_intersection point;
            control_intersection( bool boolean): intersectionBoolean(boolean),
point(0,0,0,0){}
```

```
            control_intersection( bool boolean, control_entry_intersection ce):
intersectionBoolean(boolean), point(ce){}
        };
        void newNeighbour(control_entry_neighbour);

        bool checkDistance(control_entry_neighbour);
        control_intersection checkTimeIntersectionPoint(control_entry_neighbour);

        void updateNeighbourTable();
        void updateDistanceTable();
        void updateIntersectionTable();

        void clearNeighbourTable();
        void clearDistanceTable();
        void clearIntersectionTable();

        void printNeighbourTable();
        void printDistanceTable();
        void printIntersectionTable();


};
```

controller.cpp

```
#ifndef CONTROLLER_H
#define CONTROLLER_H
#include "controller.h"
#endif
#include <stdlib.h>     /* abs */
#include <iostream>
#include <iomanip>


using namespace std;
 using namespace std::chrono;
controller::controller( vanetza::PositionProvider&
positioning):positioning_(positioning){
    TIME_LIMIT_NEIGHBOUR_TABLE = 2000000;//miliseconds
    TIME_LIMIT_INTERSECTION_TABLE = 10;//seconds
    TIME_INTERVAL_INTERSECTION = 2;//seconds
    BEARING_LIMIT = 10;
    TIME_LIMIT_DISTANCE_TABLE = 2000000;//milliseconds
    SAVE_INTERSECTION_TABLE = true;
    NEIGHBOURS_HEADING_AVAILABLE = true;
    OWN_HEADING_AVAILABLE = true;

};

void controller::updateValues(){
    auto position = positioning_.position_fix_complete();
    if(OWN_HEADING_AVAILABLE){
        bearing_ = position.course.value();
    }else{
        bearing_ =
calculateBearing(position.latitude.value(),position.longitude.value());
    }

    latitude_ = position.latitude.value();
    longitude_ = position.longitude.value();
    speed_ = position.speed.value();

}

double controller::calculateBearing(double newLatitude, double newLongitude ){

    auxiliar_gps::simple_point previousPoint = auxiliar_gps::simple_point(latitude_,
longitude_);
    auxiliar_gps::simple_point newPoint = auxiliar_gps::simple_point(newLatitude,
newLongitude);
```

```
    double heading = auxiliar_gps::bearingBetweenPoints(previousPoint, newPoint);

    return heading;

}

void controller::newNeighbour(control_entry_neighbour neighbour){

    updateValues();

    controller::control_intersection checkIntersection(false);
    if(!NEIGHBOURS_HEADING_AVAILABLE){
         neighbours_.checkAndAddEntry( neighbour);
    }
    if (controller::checkDistance(neighbour)){
        checkIntersection = controller::checkTimeIntersectionPoint(neighbour);
    }

    if(checkIntersection.intersectionBoolean){
        cout << "CHOQUE:";
        checkIntersection.point.print();
    }

    if(SAVE_INTERSECTION_TABLE){
        updateIntersectionTable();
    }

    updateNeighbourTable();
    updateDistanceTable();

}

bool controller::checkDistance(control_entry_neighbour neighbour){

    auxiliar_gps::simple_point currentSimplePoint =
auxiliar_gps::simple_point(latitude_,longitude_);
    auxiliar_gps::simple_point neighbourSimplePoint =
auxiliar_gps::simple_point(neighbour.getLatitude(), neighbour.getLongitude());

    double distance = auxiliar_gps::distance(currentSimplePoint, neighbourSimplePoint);

    return distances_.checkAndAddEntry(neighbour.getId(), distance,
neighbour.getTime());

}
```

```
controller::control_intersection
controller::checkTimeIntersectionPoint(control_entry_neighbour neighbour){

    double distance, time_intersection, my_time_to_intersection;
    bool isIntersection;
    auxiliar_gps::simple_point intersection;

    if( abs(neighbour.getHeading() - bearing_) < BEARING_LIMIT){//Two mobiles in same
directions

        auxiliar_gps::bearing_point currentPoint =
auxiliar_gps::bearing_point(latitude_, longitude_, bearing_);
        auxiliar_gps::bearing_point neighbourPoint =
auxiliar_gps::bearing_point(neighbour.getLatitude(), neighbour.getLongitude(),
neighbour.getHeading());

        if(auxiliar_gps::getBackPoint(currentPoint, neighbourPoint)){//I'm back

            if(speed_ > neighbour.getSpeed()){//Faster than the one in front, possible
collision

                isIntersection = true;
                distance = distances_.getDistanceEntry(neighbour.getId());
                time_intersection = distance/(speed_ - neighbour.getSpeed());
                my_time_to_intersection = time_intersection;

                intersection = auxiliar_gps::pointFollowingBearing(currentPoint,
distance);

            }else{
                isIntersection = false;
            }


        }else{//I'm in front

            if(neighbour.getSpeed() > speed_){//Slower than the one in back, posssible
collision
                isIntersection = true;
                distance = distances_.getDistanceEntry(neighbour.getId());
                time_intersection = distance/(neighbour.getSpeed() - speed_);
                my_time_to_intersection = time_intersection;

                intersection = auxiliar_gps::pointFollowingBearing(neighbourPoint,
distance);

            }else{
                isIntersection = false;
```

```
            }
        }

    }else{//Two mobiles in different directions
        auxiliar_gps::bearing_point currentBearingPoint =
auxiliar_gps::bearing_point(latitude_,longitude_,bearing_);
        auxiliar_gps::bearing_point neighbourBearingPoint =
auxiliar_gps::bearing_point(neighbour.getLatitude(), neighbour.getLongitude(),
neighbour.getHeading());

        intersection = auxiliar_gps::intersection(currentBearingPoint,
neighbourBearingPoint);

        auxiliar_gps::speed_point currentSpeedPoint =
auxiliar_gps::speed_point(latitude_, longitude_, speed_);
        my_time_to_intersection = auxiliar_gps::timeToPoint(currentSpeedPoint,
intersection);

        auxiliar_gps::speed_point neighbourSpeedPoint =
auxiliar_gps::speed_point(neighbour.getLatitude(), neighbour.getLongitude(),
neighbour.getSpeed());
        double neighbour_time_to_point = auxiliar_gps::timeToPoint(neighbourSpeedPoint,
intersection);

        double interval_intersection = abs(my_time_to_intersection-
neighbour_time_to_point);

        if(interval_intersection < TIME_INTERVAL_INTERSECTION ){
            isIntersection = true;
        }else{
            isIntersection = false;
        }
    }

    if(isIntersection && time_intersection < TIME_LIMIT_INTERSECTION_TABLE ){
        std::cout << neighbour.getId()<< std::endl;;
        milliseconds timeNow = duration_cast< milliseconds
>(system_clock::now().time_since_epoch());

        control_entry_intersection entry_intersection =
control_entry_intersection(neighbour.getId(),

timeNow.count() + my_time_to_intersection * 1000,

intersection.latitude,

intersection.longitude);
        if(SAVE_INTERSECTION_TABLE){
```

```
            intersections_.checkAndAddEntry(entry_intersection);
        }

        controller::control_intersection out = controller::control_intersection(true,
entry_intersection);
        return out;

    }else{

        controller::control_intersection out = controller::control_intersection(false);
        return out;

    }


}

void controller::updateNeighbourTable(){
    neighbours_.deleteByTime( TIME_LIMIT_NEIGHBOUR_TABLE);
}

void controller::updateDistanceTable(){
    distances_.deleteByTime(TIME_LIMIT_DISTANCE_TABLE);
}

void controller::updateIntersectionTable(){
        intersections_.deleteByTime();
}

void controller::clearNeighbourTable(){
    neighbours_.clearTable();
}

void controller::clearDistanceTable(){
    distances_.clearTable();
}

void controller::clearIntersectionTable(){
    intersections_.clearTable();
}


void controller::printNeighbourTable(){
    neighbours_.print();
}

void controller::printDistanceTable(){
    distances_.print();
```

```
}

void controller::printIntersectionTable(){
    intersections_.print();
}
```